27654

NATIONAL LIBRARY
OTTAWA

BIBLIOTHÈQUE NATIONALE
OTTAWA

CANADA

NAME OF AUTHOR ... Christopher Gray ...

TITLE OF THESIS ... ALAI: a Language for Intelligence ...

UNIVERSITY ... University of Alberta ...

DEGREE FOR WHICH THESIS WAS PRESENTED ... M. Sc. ...

YEAR THIS DEGREE GRANTED ... 1976 ...

Permission is hereby granted to THE NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) ... Christopher Gray

PERMANENT ADDRESS:

#1612 - 8515 - 112 St
Edmonton, Alta.
T6G 1K7

DATED March 3 ... 1976

NL-91 (10-68)

INFORMATION TO USERS

THIS DISSERTATION HAS BEEN
MICROFILMED EXACTLY AS RECEIVED

This copy was produced from a microfiche copy of the original document. The quality of the copy is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

PLEASE NOTE: Some pages may have indistinct print. Filmed as received.

Canadian Theses Division
Cataloguing Branch
National Library of Canada
Ottawa, Canada K1A 0N4

AVIS AUX USAGERS

LA THESE A ETE MICROFILMEE
TELLE QUE NOUS L'AVONS RECUE

Cette copie a été faite à partir d'une microfiche du document original. La qualité de la copie dépend grandement de la qualité de la thèse soumise pour le microfimage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

NOTA BENE: La qualité d'impression de certaines pages peut laisser à désirer. Microfilmée telle que nous l'avons reçue.

Division des thèses canadiennes
Direction du catalogage
Bibliothèque nationale du Canada
Ottawa, Canada K1A 0N4

THE UNIVERSITY OF ALBERTA

ALAI: A LANGUAGE FOR ARTIFICIAL INTELLIGENCE

by

CHRISTOPHER GRAY

©

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE
OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTING SCIENCE

EDMONTON, ALBERTA
SPRING, 1976

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled "ALAI: A Language for Artificial Intelligence", submitted by Christopher Gray in partial fulfillment of the requirements for the degree of Master of Science.

....
Co-Supervisor

L.K. Schubert
....
Co-Supervisor

Barry J. Mailloux
....

Dale H Bent
....

Dated Feb. 23 .... 1976.

Abstract

The use of computers in artificial intelligence applications benefits from a programming language suited to the task. Many such languages have been developed recently, but programs written in most of them cannot readily be transported or combined with programs written in the others. Total effort could be reduced and total results increased, if the various projects could be brought together and merged. This requires the existence of a single standard AI programming language which combines the facilities and capabilities of the languages previously used for the various individual projects.

In this thesis, several programming languages are discussed with respect to their applicability to AI programs. Their useful features are extracted and a language combining these and other features is proposed. The proposed language (ALAI) is described in detail and examples are provided to clarify the intended usage. Some of the features stressed in the language are easy iteration, pattern matching, data net manipulation and program modifiability.

The language contains several features which have not been used in previous languages. These features are intended to improve the efficiency of data net search and internal program environment changes. The use of backtracking as a problem solving tool is discussed and ways in which the process can be speeded up are introduced.

Acknowledgments

I would like to thank Dr. Jeff Sampson and Dr. Len Schubert for their continual advice, support and criticism which has helped shape this thesis and the ALAI language.

I would also like to thank Jim Heifetz for his many discussions, criticisms and suggestions regarding the language.

I wish to extend my appreciation to the Department of Computing Science and to the National Research Council for their support of this work.

CONTENTS

CHAPTER PAGE

1. Introduction 1

2. Other AI Languages 4

   2.1 LISP 5
   2.2 SAIL 11
   2.3 SNOBOL 17
   2.4 2.PAK 23

3. The Features of ALAI 30

   3.1 Basic Needs 32
   3.2 Basic Internal Forms 34
   3.3 Portability and Redundancy 37
   3.4 The Data Net 39
   3.5 Pattern Matching 44
   3.6 Backtracking 47
   3.7 Planning 54

4. The Language 55

   4.0 Method of Description 55
   4.1 Basic Structure 55
   4.2 Identifiers 59
   4.3 Declarations 60
   4.4 Expressions 69
   4.5 Statements 76
   4.6 Arithmetic 78
   4.7 Strings 81
   4.8 Bits 84
   4.9 Booleans 86
   4.10 Pairs, Triples and Vectors 89
   4.11 Sets, Bags, Lists and Chains 92
   4.12 Stacks and Queues 97
   4.13 Records 100
   4.14 User Defined Data Types 103
   4.15 Labels 105
   4.16 Procvals 106
   4.17 Locations (loc's) 107
   4.18 Arrays 108
   4.19 Tables 110
   4.20 Areas 112
   4.21 Conditional Constructs 112
   4.22 Iterative Constructs 115
   4.23 Return, Exit, Iterate and Next 121

CONTENTS (cont'd)

CHAPTER PAGE

4.24 Input and Output 122
4.25 The Data Net 126
4.26 Unevaluated Expressions and Parse Time and Compile Time Evaluation 132
4.27 Patterns and Pattern Matching 134
4.28 Contexts 142
4.29 Processes 146
4.30 Events and Event_types 152
4.31 Backtracking 156
4.32 Worlds 166
4.33 Elaborations 170
4.34 Odds and Ends 174

5. Conclusion 177

* * *

References 182

Appendix A - A Grammar for ALAI 183

Appendix B - Internal Structures 193
B1 Language Defined Record Classes 193
B2 Internal Program Structures 197
B3 Symbol Tables 201

Appendix C - Operating Environment 204

Appendix D - Sample Programs 206

CHAPTER 1

INTRODUCTION

Computers, often thought of as machines which compute, have many uses other than as large, high speed numerical calculators. One of these uses is as a tool for the exploration of intelligence, i.e. finding out what intelligence is, what it encompasses and how it is achieved. The main use of computers in such studies is as a medium in which intelligent processes can be simulated, a usage embodying the field usually termed "artificial intelligence". This field covers such topics as computerized game playing, automatic theorem proving, the understanding and production of natural language by machine, simulation of human thought, etc. Programming computers to accomplish such tasks is very difficult, partly because of the innate complexity and abstractness of the tasks. Special programming languages and techniques have been developed which can make the job easier. This thesis discusses some of these languages and techniques and introduces a new language which provides a combination of techniques which has not previously been directly available.

Chapter 2 discusses several existing programming languages which have been used in artificial intelligence

(AI) applications. The particular languages discussed were chosen because of the diversity of their characteristics and the influence they have had on the proposed new language. The discussion concentrates on the language features which most strongly affect the ease with which the language under discussion can be used to program a major AI application. Mention is made of the strong and weak points of the languages and, where relevant and possible, comparison is made between the languages as to the usability of the forms in which the capabilities in common are offered.

Chapter 3 discusses the possibility and desirability of combining many language features (including those discussed in Chapter 2) in a single AI language. Consideration is given to what features are desired and how they can be combined so as to minimize inconvenience and effort on the part of the programmer. The language ALAI (pronounced al-eye) is introduced as an attempt to provide a previously unavailable combination of features and capabilities. The chapter includes motivations for some of the features included in the new language and continues the comparison of features and ways of attaining them. Alternate forms which are not used in ALAI are discussed, along with reasons why they are not used. The major topics are the data network, pattern matching, backtracking and processes.

The central part of the thesis is Chapter 4, which describes the proposed language in detail. The description is arranged in sections according to data types and basic language constructs. The base constructs, i.e. declarations, statements and expressions, are described first, after which the features are described in order of increasing complexity or unfamiliarity to the average reader. The descriptions deal mostly with the semantics, i.e. meaning and usage, of the types and constructs, while the syntactic form is described by a grammar in Appendix A. Example uses of the features are given along with explanations where necessary.

CHAPTER 2

OTHER AI LANGUAGES

In the past few years programming languages intended for or usable in artificial intelligence applications have been appearing quite rapidly. Those responsible for the new languages feel that existing ones are inadequate. FORTRAN, ALGOL, COBOL, etc. do not meet the needs of most current AI programmers. Projects in AI are becoming increasingly ambitious and varied. They range from game playing and theorem proving to natural language information retrieval and generalized problem solving. Attempting to instruct computers (notoriously unimaginative and rigid) to solve such problems is difficult even without the often severe limitations of conventional programming languages and their inconsistent implementations. The task becomes easier with the use of a programming language suited to the specialized needs. Many of the new languages are of interest because of their greater utility in AI applications and also because of the intrinsic interest of the new programming constructs and data handling concepts which they introduce.

Before attempting to design a new programming language it is useful to examine the existing languages in order to learn from the mistakes and successes of others. A detailed study of all the existing AI languages, their merits, shortcomings and structure is beyond the scope of this thesis. The discussion will consider only four languages. These four, LISP (Weissman, 1967; Smith, 1970), SAIL (Van Lehn, 1973), SNOBOL (Griswold, et al., 1971) and 2.PAK (Melli, 1974) were chosen because of their influence on the present work and because of the diversity of their formats and capabilities. Other languages which have been highly influential include PLANNER (Hewitt, 1969, 1971), CONNIVER (Sussman, et al., 1972; McDermott, et al., 1972), QA4 (Rulifson, et al., 1972), POP-2 (Burstall, et al., 1968) and ALGOL68 (Van Wijngaarden and Mailloux, et. al., in press). Brief discussions of a selection of AI languages can be found in Bobrow and Raphael, 1974.

2.1 LISP

LISP is one of the few languages that is automatically associated with artificial intelligence by most people in the computing field. This is perhaps due to the nature and form of the language, both of which tend to discourage most casual users. The unusualness of the language stems from its completely uniform representation scheme for programs, data, system constructs, etc. and from its surface appearance which is that of nested lists enclosed in brackets. Newer forms of the language accept input in more standard forms, e.g. MLISP looks like an ALGOL language. For this discussion, the original LISP format will be considered. The surface form has only four special characters (disregarding those which serve only as "break" characters and "macros" for some syntactic structure), '.', '(', ')' and the space. Thus input to and output from the system are simple and uniform, but because of the great nesting depths commonly encountered, the structures can be very complex and difficult to understand. Also, because such heavy use is made of the few special characters, it is very easy to misuse them, the most common problem being that of mismatched or misplaced parentheses.

The base entity in LISP is the atom, which is represented via its name, a sequence of characters. At the next level is the dotted pair, represented as '(', the left field, '.', the right field and ')'. The fields can be either atoms or other dotted pairs. Thus all things in LISP are atoms or binary trees created from atoms. (Here "tree" is taken to allow looping structures, i.e. a branch of the tree can point back to the root, etc.) Linked lists can be represented as dotted pairs by letting the left fields be the elements of the list and the right fields be the links which define the list. E.g. (A . (B . (C . NIL))) is a linked list with the three elements A, B and C. Such forms are so common that a special syntax (list notation) has been designed for them: (A B C), etc. This is a surface convention only; internally, lists are still represented by right branching chains of dotted pairs.

Programs, expressions and commands are represented by functions, predicates, pseudo-functions and special forms, all of which are in turn represented as lists, the first element of which specifies the function, etc. to be used and the remainder of which specifies the arguments (each argument and the function are evaluated before the function is called). E.g. (PLUS 3 (TIMES 2 7)) evaluates to 17. All functions, special forms, etc. yield a value when executed, hence they can all be used as expressions. Since all structures are binary tree structures, the only capabilities really needed are those for setting the left and right fields of dotted pairs (REPLACA and REPLACD), for examining these fields (traversing the tree to the left or right respectively) (CAR and CDR) and for some kind of conditional capability. Others are usually provided, however, to simplify list manipulations, function definition, program control, etc. It is often desirable to simply build lists, rather than execute them, i.e. to be able to use a form such as (REPLACD location (A B C)) without having the system try to execute function A with arguments B and C. This is provided for by the special form QUOTE which simply returns its argument, unevaluated.

In keeping with the mathematical consistency and simplicity of LISP structures, a form of the Church lambda notation is used for the definition of user functions. Such forms can be named and hence saved in the system or can be unnamed "constants" which exist only for their one usage.

The form is illustrated, along with the basic LISP conditional, in the following factorial function:

```
(DEFINE (QUOTE (
(FACTORIAL (LAMBDA (N)
(COND ((ZEROP N) 1)
      (T (TIMES N (FACTORIAL (DIFFERENCE N 1))))))
)) )))
```

DEFINE takes as argument a list of function definitions. Each function definition is a list containing the atom representing the name and a lambda expression representing the body of the function. A lambda expression is 'LAMBDA' followed by an (ordered) list of parameters, followed by an expression which usually references the parameters. The COND special form takes as arguments several two element lists. It successively evaluates and tests the results of the first element of these lists. If the result is not the special atom NIL, then the value of the COND is the value of the second part of that list. The atom T is used to correspond to 'true' as NIL corresponds to 'false'. ZEROP is non-NIL if its argument is zero. Thus this formulation of factorial can be read as "define the function FACTORIAL such that FACTORIAL(N) is given by: if N is zero then 1 else N times FACTORIAL(N-1)".

The above definition can be executed at any time, e.g. as part of the execution of some other program. A DEFINE which refers to an atom which already has an associated function simply redefines the function. Associated with each atom is a property or p-list which contains indicators, optionally followed by values, which constitute knowledge concerning that atom. Functions are simply stored on this list as the value of some indicator such as EXPR. The user is free to manipulate p-lists in any way he sees fit, thus he can easily produce invalid functions, possibly causing LISP to fail. Also, since such functions are list structures, they can be changed and created by other programs. A program can even dynamically modify itself.

These features have several implications for AI programmers. The surface syntax of LISP can be very complex, difficult to read and very verbose. For example, compare the above factorial definition with the following ALGOLW version:

```
INTEGER PROCEDURE FACTORIAL(INTEGER VALUE N);
   IF N=0 THEN 1 ELSE N*FACTORIAL(N-1);
```

This version is slightly shorter and, to most people, much clearer. Also, the syntax used, although it incorporates several different rules, is more pleasing in that, once learned, it can more readily be written without error.

The program/data uniformity gives LISP programs the capacity for self improvement of a scale and nature not readily possible by simple parameter optimization. This capacity, usually considered necessary for truly ambitious attempts at machine intelligence, is simply not available in standard programming languages. A uniform program representation would seem to be desirable, at least internally, so that self modifying systems need not be overly concerned with syntax. The LISP convention does not entirely eliminate this problem, however, since it is still necessary to keep track of how many of what kind of arguments the various functions, predicates, etc. require.

The uniform binary tree structure used for data also has its advantages and disadvantages. Its simplicity and uniformity allow the design of powerful functions which operate on such structures. Their purpose and method of operation is less hidden in details of the data structure than the purposes and methods of functions which must deal with complicated structures having many types of components. Also, the constant size (two pointers) of all nodes, records, etc. used by the system, greatly simplifies storage management. This same uniform representation, however, tends to obscure non-binary properties by requiring that nodes using more than two fields be represented as lists or some other structure built up from two field nodes (dotted pairs). Also, the list representation can be wasteful of space. For example, the list (DIFFERENCE X Y) uses three dotted pairs for a total of six pointers. If a three field node were used, however, (DIFFERENCE is known to always require exactly two arguments, hence examining the first field can give the total number of fields) only three pointers would be needed. Thus LISP provides several features which are highly desirable for AI applications, but makes some sacrifices to do so.

2.2 SAIL

The SAIL language is almost totally dissimilar from basic LISP. SAIL, and its partner the LEAP data structures, are easy to use and are efficient in terms of computation. The language is ALGOL based and can be compiled to produce object modules which can be saved and then loaded and executed with other programs (possibly written in some other language) at a later date. Close control over input and output is provided. SAIL has several data types, including associations, items, sets and lists. Syntactic constructs are provided for easy manipulation of these types and for many standard operations found in other programming languages (e.g. integer and real arithmetic, arrays, automatic type conversions, etc.). The basic language is similar to other ALGOL type languages and need not be detailed here. Of special note is the extended capability of the iteration statement as in:

```
FOR X<-2, I-6, J STEP 3 UNTIL K, L STEP M-2 WHILE P<27 DO
    statement;
```

Here X will take on successive values of 2, I-6, J, J+3, ... K, L, L+M-2, ... etc. and the 'statement' will be executed once for each of these values. DONE, NEXT and CONTINUE statements are provided for exiting and iterating such loops.

Of chief interest to AI workers are the sets, lists and data structure provided, and the facilities for manipulating them. A list is an ordered sequence of elements from which individual elements can be picked out by indexing. Elements can be inserted (at specific locations) or deleted via the PUT and REMOVE statements. The order of occurrence and the number of occurrences of an element in a list is wholly determined by the user program; SAIL will not change them. Sets are like lists except that the user programs cannot control the ordering of elements within the sets and an element can occur only once; SAIL sets correspond to the standard mathematical notion of a set. The operations of set union, intersection and difference are provided. An additional iteration statement, the FOREACH statement, is introduced. It is capable of iterating through a set or list, i.e. of doing some statement once for each element (the current value of the iteration variable) of the set or list.

The SAIL data structure is built up of items. Each item can have an associated PNAME or print name and can have a datum, which is any arbitrary value, associated with it. Internally, items are represented by integers; each item has a corresponding integer which is globally reserved for that item (all parts of the data structure are outside of the block structure of SAIL programs). The data structure consists of associations, three element entities representing knowledge concerning items. The three elements are the attribute, the object and the value. For example Father-of⊗John≡Tom can be read as "the father of John is Tom". The attribute, object and value are the items

Father-of, John and Tom respectively. Other associations can be used as parts of associations, thus knowledge about knowledge is possible. Such associations can be made and unmade via the MAKE and ERASE statements.

Searching of and retrieval from the SAIL data structures (LEAP) are greatly facilitated by the FOREACH statement used in conjunction with derived sets. For example, consider the following:

```
FOREACH X | FATHER⊗TOM≡X AND X IN PTA-SET DO statement
```

The 'statement' will be done once for each FATHER of TOM who is a member of PTA-SET. If a derived set is used, the generality increases:

```
FOREACH X,Y | LINK⊗(FATHER⊗Y)≡X DO ...
```

This FOREACH will iterate through all LINKs of all FATHERs. The derived set is (FATHER⊗Y). The three forms for such derived sets are as follows: a⊗b represents the set of all X such that a⊗b≡X, a≡b represents the set of all X such that X⊗a≡b, and a'b represents the set of all X such that a⊗X≡b.

The internal nature of SAIL's data structure is also of interest. Many languages with a built-in data structure or data net have it in the form of linked lists. These lists must be traversed in order to locate information on them. This time consuming procedure is circumvented in SAIL. Since each association has exactly three components, it is possible to store the associations in tables. The position of a particular association can be determined by hashing on the integer values of the items used. The elements of derived sets can be obtained by following short, specialized linked lists containing only the relevant values. To symmetrically allow the three types of derived sets, all associations are stored in triplicate, hashed and linked in three different ways. In each of the three tables, associations having the same elements in two positions are stored close together (partly due to hashing, but mostly due to a paging technique) so that iteration through a derived set can, operating system permitting, involve a minimum number of references to bulk storage devices (drums, disks, etc.). The SAIL data structures can provide significant improvements in search and retrieval times, but they suffer from redundancy, inefficient use of storage and an often cumbersome restriction to three element associations.

SAIL allows multiple concurrent processes via timeslicing. The various processes can communicate with each other and with the outside world via events of specific event types. A given process can be in any of four states: terminated, suspended, ready or running. Processes are created by the SPROUT statement, suspended and terminated by the SUSPEND and TERMINATE statements, and can be made ready (resumed) by the RESUME statement. A process is running only when it is the one actually executing (through the timeslicing scheme) at that time. Ready processes are conceptually running, but, due to the non-parallel nature of typical computers, they are not actually executing.

Suspended processes are non-ready but can be made ready by a RESUME in the running process. RESUME provides a means by which the issuer of the RESUME can pass a value (a message) to the process being resumed. Terminated processes have been permanently discontinued and can never be made ready or running again.

Events in SAIL are arranged into event types. Each event type has an associated queue of processes waiting for an event of that type (the wait queue) and an associated queue of events of that type waiting to be noticed (the notice queue). The CAUSE statement produces events and can thus affect notice queues and can cause the resumption of processes waiting for an event of the type caused. The INTERROGATE form examines the queues of some event type and can affect both the event type and the running processes. The JOIN construct (similar to the waiting action of INTERROGATE) causes one process to be suspended until all of a specified group of processes are terminated. Each event carries with it an associated message which can be used to identify the event and carry semantic information about it. Provisions are made whereby the user can write his own procedures for CAUSING and INTERROGATING event types.

SAIL also provides a context mechanism of the "storage bin" variety. Individual variables or all variables previously remembered in that context can have their current value saved in the context via the REMEMBER statement. It also allows the remembering in one context of the current values of the variables which have been stored in another context. The RESTORE (for retrieving saved values) and FORGET (for discarding saved values) statements are similar. The INCONTEXT form tests whether or not a given variable is saved in a given context.

This type of context mechanism gives the programmer a method for storing information about states of the world in such a way that he has complete control over what is saved where. It does not, however, offer any facilities whereby variables (or array elements or item data, etc.) can be flagged so as to be automatically saved when they are changed. Thus it is necessary to explicitly save everything that needs to be saved, a task which can be difficult if the structure, as well as the content of a world model is changing over time. SAIL also offers no means whereby a program can easily create other programs which are equivalent to itself in terms of accessibility for running. The overall structure of the language suffers from some rigidity (especially in relation to the data structure) which can make some applications cumbersome and inelegant.

2.3 SNOBOL

The SNOBOL language (the SNOBOL4 language as described by Griswold, et al., 1971) is similar to LISP in that the external and internal forms are not like those of standard programming languages. All SNOBOL statements have the same basic form, which is similar to that used by assembly language in that it is divided into specific fields. The general form is LABEL SUBJECT PATTERN = OBJECT GOTO. The LABEL is used to name the statement so that it can be branched to; if it is not present, then at least one blank must precede the SUBJECT. The GOTO is indicated by a colon prefix and, if not present, defaults to an unconditional branch to the next statement.

The sole decision making capability in SNOBOL is that of success or failure of an operation. If any operation in a statement fails (e.g. array index out of bounds, a predicate does not hold, pattern matching fails, etc.) then the execution of the statement immediately ceases and the statement is said to have failed. The three forms of GOTO, ':S(label)', ':F(label)' and ':(label)' control branching on success, on failure and unconditionally. The success and failure form can appear together. If the PATTERN does not appear in a statement, then the statement is just a simple assignment statement (one can think of an omitted pattern as matching any SUBJECT) as in

```
   COUNTER = COUNTER + 1
```

If both the = and the OBJECT are omitted, the statement is a pattern matching statement in which the PATTERN is matched against the SUBJECT. If all of SUBJECT, PATTERN, = and OBJECT are present, then the statement is a replacement statement in which the substring of SUBJECT which was matched by PATTERN is replaced by OBJECT. If OBJECT is omitted but the = is present, then OBJECT is assumed to be null (a string of length zero) for either an assignment statement or a replacement statement. Any other statement (except the end statement, which has only the label END), such as LABEL = OBJECT, etc. is illegal.

The internal workings of SNOBOL are not accessible to the programmer, but some hint can be obtained from the definition of the language. Each string is represented uniquely inside the system so that if we have A = 'BOTHSTRINGS' and B = 'BOTH' 'STRINGS' (the blank represents string concatenation) then both A and B point to the same internal location. Each string has associated with it a value which can be changed and referenced by the user. For example, if we have

```
   MONTH = 'APRIL'  .
```

then $MONTH is equivalent to APRIL. E.g.

```
   $MONTH = 'CRUEL'
   MEAN = $MONTH  .
```

are equivalent to

```
   APRIL = 'CRUEL'
   MEAN = APRIL
```

The $ indirection operator can of course be applied to string expressions as in:

```
WORD = 'RUN'
$(WORD 'HOME') = $(WORD 'HOME') + 1
```

which increment the value of RUNHOME. This equivalence of string expressions and compile time variables via the $ operator is an interesting feature of the language. It allows such things as computed goto's:

```
N = N + 1         :($('LABEL' N))
```

which branches to LABELn depending on the value of N.

Perhaps the most interesting feature of SNOBOL is its pattern matching capability. A pattern can be thought of as a set of strings. A pattern match is said to succeed if some substring of the SUBJECT is equal to one of the strings represented by the PATTERN. Basically, patterns are constructed from string constants, variables (which represent the string form of their current value), the alternation operator (|) and the concatenation operator (blank). Brackets can be used to alter the order of application of the operators (concatenation has higher precedence). Thus if

```
PUSS = 'CAT'
```

then the pattern

```
PUSS ('KIN' | 'TAILS' | 'ERPILLAR')
```

represents the strings 'CATKIN', 'CATTAILS' and 'CATERPILLAR'. The value of a variable used in a pattern can be another pattern which is then used as a subpattern of the pattern in which the variable appears.

A number of special purpose built-in patterns are also available. NULL represents a string of length zero. LEN(integer) represents a string with (integer) characters. SPAN(string) and BREAK(string) match, respectively, sequences of, and anything up to, characters which are in (string). ANY(string) (NOTANY(string)) matches single characters which are (are not) in (string). TAB(integer) and RTAB(integer) move the cursor (the pointer indicating the current position in the subject string) to the (integer) position from the left or right end of the string. Left movement (backwards) is not permitted for TAB and RTAB. REM is equivalent to RTAB(0), it matches the REMainder of the string. POS(integer) (RPOS(integer)) fails if the cursor is not at the (integer) character from the left (right) end of the subject string. FAIL causes a fail (in the pattern matching) when encountered. FENCE matches the null string when initially matched, but causes failure of the entire pattern match if backed into. ABORT causes immediate failure of the entire pattern match. ARB matches any string of zero or more characters, but will always attempt to match as few characters as possible. ARBNO(pattern) matches any number (as few as possible) occurrences of (pattern). BAL matches a string representing an expression which is balanced with repect to parentheses. SUCCEED matches the null string and will always succeed.

Several other features of patterns are also available.

If a pattern is postfixed by . and a variable, then if the match succeeds, the variable is assigned the portion of the subject which was matched by the postfixed pattern. $ is used in this way like . except that the assignment is done during the pattern matching and does not depend on overall success. The character @ immediately before a variable (no intervening blanks as in . or $ and other SNOBOL operators) produces a pattern which matches the null string and which immediately assigns the current cursor position (an integer value) to the variable. When a pattern is produced, any expressions in the pattern are normally evaluated immediately. Thus when actual pattern matching occurs, all parts of the pattern are constants. The evaluation of expressions can be delayed until pattern matching time by prefixing the expression with *. This use of unevaluated expressions allows the construction of recursive patterns as in

```
   P = *P 'Z' | 'Y'
```

which matches a 'Y' followed by zero or more 'Z's. SNOBOL also contains a number of user settable flags which control some of the details of pattern matching, input/output, program tracing, etc.

SNOBOL also provides many system functions, predicates, operators, etc. which are needed for basic programming capability. One feature not often found is the ability to redefine the meaning of an operator or function name and to define synonyms. For example OPSYN('+','F',2) causes all subsequent uses of the operator '+', as in N + M to call the function F, as F(N,M). As in LISP, the definition of a function is an executable operation, so that functions can be dynamically defined and redefined. An interesting data structure is the table, which is essentially a one dimensional array which is indexed by anything, rather than just by integers. Thus if T is a table, T<'A'> = 3 assigns 3 to the A'th element of T. A is not converted to an integer, it is simply a string value. Similarly, patterns, real numbers, unevaluated expressions, etc. can be used to index tables. SNOBOL provides a way whereby executing programs can modify themselves, but the method is far less elegant and complete than that of LISP. The function CODE accepts a string argument which is compiled into executable internal structures. Labels in the new code supercede existing labels of the same name. A GOTO of the form :<code> will cause a branch to the top of the program pointed to by (code).

The language provides excellent pattern matching capabilities as far as strings are concerned, but does not extend these capabilities to other data types such as integers or user defined data types (similar in SNOBOL to records offered by typical ALGOL languages). No facilities are provided whereby a data network can be created, manipulated or saved for future use (without cumbersome use of user data types, string pattern matching, external storage, etc.). Some rather arbitrary restrictions exist in the language; for example, an array may have another array as an element, but a form like A<12,6><14,J> may not index the inner array; instead, the system fuction ITEM must be used. Similarly, ($S)<2> is not allowed. The external format of the language is useful for certain restricted applications, but it tends to be difficult to get used to and can be difficult to follow. Also, the control structures offered are somewhat restricted and the power and convenience of iteration is totally lacking.

2.4 2.PAK

2.PAK, like SAIL, is an ALGOL type language, and as such it is relatively easy to use. One of the most noticeable features of 2.PAK is its conciseness; the constructs available are few and do not overlap much. The programmer is thus not confused by a bewildering array of special forms and methods, many of which effectively do the same thing. The language provides the very basic capabilities for working with the features offered, but leaves the definition of more powerful, more specific forms up to the user (via macros and procedures). This convention is good for programs which use the various features in straightforward ways, but more complicated programs may tend to become overfull of calls to procedures and macros which have functions known for certain only by the programmer. A price of basic simplicity and specificity can thus be inconsistency.

The use of macros in higher level languages is not new, but is sufficiently rare to be noteworthy. The macro facility of 2.PAK is by no means as extensive as the macro and conditional assembly features of the IBM/360 assembler, but is sufficient for most needs. Parameters are allowed, but they are strictly ordered and are only string constants or identifiers. Expressions and variables in 2.PAK are typed as fully as possible (e.g. 'string list array') so that the compiler can aid the programmer by type checking his programs. For applications requiring variable types, the all inclusive type 'var' is provided along with execution time type determination via the procedure 'type'. Like SNOBOL, 2.PAK features tables (typed as to the entries) and unevaluated expressions. The prefix '#' is provided to force evaluation within an unevaluted expression.

The coroutines provided by 2.PAK seem to be a cross between generators (as in CONNIVER) and processes (as in SAIL). They are declared like procedures and appear identical to them except that they cannot return a value. Variables can be declared which can point only to individual instances (invocations) of some specific coroutine. Values for these are created by calling the coroutine with some specific arguments. Each such call creates a new and independent coroutine instance. Three statements are provided for manipulating coroutine instances. The 'invoke' statement causes the specified instance to be restarted at the point at which it previously left off. The instance issuing the 'invoke' is suspended and the return location of the invoked instance is set to the statement following the 'invoke'. The 'resume' statement is like 'invoke' except that the return location of the instance being resumed is not affected. The 'detach' statement causes the current instance (the one issuing the 'detach') to suspend itself and branch to its return location. Each coroutine contains an implicit 'detach' as its final statement. 'Detach' and 'resume' can occur only in coroutines. The reserved word 'me' points to the currently active coroutine instance. Variables within a coroutine instance can be referenced, both for retrieving and changing their value by preceding the variable name (or array, list or table element) with the variable pointing to the instance followed by a period. This convention is recursive, allowing such forms as X.Z.P(6,4):=2.

The data network offered by 2.PAK is based on labelled nodes interconnected by labelled, directed edges. It is constructed using the standard language constructs and record classes. The entire package could have been programmed by the user, but is provided as part of the language to simplify the user's job and to provide some uniformity among 2.PAK programs. Pattern oriented searching of and retrieval from the structures can be provided by the pattern matcher.

A type of backtracking is provided through an expanded "storage bin" context mechanism. Contexts are created and entered via the 'new context' statement. Individual values are saved in the current context by the 'save' statement. An optional argument to the data-base procedures or the setting of a global .SAVE switch allows changes to the data-base to be saved in the current context. The 'back' statement restores immediately, transfers to the earlier context, or forgets entirely changes made to variables and/or the data-base and then destroys the current context and backs up to the previous one. In this way backup of values, but not of execution, is achieved. The 'preserve' statement binds the current context to the current coroutine instance so that it can be specifically referred to. The 'restore' statement restores the world to the state it had when the context bound to the specified coroutine instance was last active. This methodology allows no way for referencing values saved in a context not bound to a coroutine (only one can be bound to each coroutine instance). Also, as in SAIL, values to be saved must always be explicitly saved by the programmer.

The pattern matcher of 2.PAK is one of the language's most interesting features. Unlike the matchers of SNOBOL and PLANNER which are restricted to matching only strings or only the data net, the 2.PAK matcher can operate on any subject pattern whatever. Also, the user is free to write pattern matching procedures which best meet his individual requirements. A 2.PAK pattern is built up of boolean expressions which must evaluate to true in order for the subpattern they represent to be successful. The global variables SUBJECT and CURSOR point to the string being matched (SUBJECT is used only for string pattern matching) and the portion of the subject currently being examined (integer character position for strings). The operator 'or' allows alternation as with SNOBOL's '|'. Bracketed subpatterns are valid "arguments" for 'or'. The process of pattern matching is one of comparison, advancing in both the subject and the pattern, and backup in both the subject and the pattern. The subject is fixed, hence retraversing it has no effect on it and will yield the same result, but the pattern contains alternatives and expressions to be evaluated, hence backing up and going forward in it can have very definite effects.

To allow full control over these effects, direction indicators are used. If a boolean expression appears by itself, it is taken as a constant and is evaluated only when passed over in the forward direction. If it is prefixed by '<->' then it is considered to be a generator of subpatterns and is evaluated when going forward and when going backward. This action is needed for patterns equivalent to, say the SNOBOL ARBNO. To complete the set, the prefix '<-' specifies that an expression is to be evaluated and tried only when backing up. Motion in the pattern is controlled by the success or failure (true or false result) of the boolean expressions, but traversal of the subject is entirely up to the expressions of the pattern. Three global variables are used to control pattern matching. If SUCCEED is true when a new subpattern is to be tried, then the entire match stops and 'true' is returned. Similarly FAIL can cause an immediate termination with a result of 'false'. If CONTEXTS is true, then a new context is created and entered for each major pattern matching expression (each sequential part) evaluated.

Several predefined functions are provided, all of which could have been written by the user. For example, 'cursor' sets the value of CURSOR and returns 'true', 'get_cursor' retrieves the value of CURSOR and returns 'true', 'match_string' compares the string pointed at by CURSOR with its argument and, if they are equal, advances CURSOR to the end of the matched substring and returns 'true' else returns 'false', etc. The basic boolean matching procedure 'match' accepts a pattern argument, an initial value for CURSOR and possibly an initial value for SUBJECT. It returns 'true' if the match is successful. For example

```
PAT:=<:(type(CURSOR)='int'), (type(SUBJECT)='string'),
    match_string('B') or match_string('R'),
    match_string('E') or match_string('EA'),
    match_string('D') or match_string('DS'):>;
```

matches the strings 'BED', 'BEDS', 'BEAD', 'BEADS', 'RED', 'REDS', 'READ' and 'READS'. Recursive patterns are possible:

```
PAT:=<:(type(CURSOR)='node'),
    <->match_edge('ABOVE'),
    (label(CURSOR)='TABLE') or
    match(PAT,CURSOR):>;
```

PAT will succeed if CURSOR initially points to a node from which there is a sequence of zero or more edges labelled 'ABOVE' leading to a node labelled 'TABLE'.

This pattern matching system is quite general and powerful but it tends to be rather verbose and does not make pattern writing easy for the programmer. Compare the above string pattern with an equivalent SNOBOL one:

```
('B' | 'R') ('E' | 'EA') ('D' | 'DS')
```

The SNOBOL version is much shorter and clearer. 2.PAK, like SNOBOL, allows the execution time compilation of strings, but this capability is far inferior to that of LISP and LISP-based languages. The data net provided does not allow for system efficiency because of its added on nature. Also, the use of edge labels to specify the relationship holding between the participant nodes may not always be desirable. For example, it does not readily allow other than two-way relationships without artificial subrelations such as are needed for multi-element relationships in LISP. Other net conventions, although representationally equivalent, may be better suited conceptually to some applications.

CHAPTER 3

THE FEATURES OF ALAI

The previous chapter has discussed several programming languages with respect to their basic structure, the features they offer and the usefulness of their various features in artificial intelligence applications. The list of features mentioned is by no means exhaustive. Features which have been used to advantage in other languages include failure driven backtracking (both of values and execution in programs), pattern directed procedure invocation, and language extensibility. None of the languages yet available offers all of these features as built-in components. It is not clear that it is desirable, or even possible to combine them all.

If all of the desirable features (or some acceptable alternative) were available in one language, it is possible that many of the current artificial intelligence projects could be rewritten in the one language without too much difficulty. This would be a first step towards producing a single large system with truly advanced capabilities. The components for natural language comprehension, natural language generation, problem solving, inductive and deductive reasoning, planning, etc. could be separately designed, yet straightforwardly integrated. At the very least, more widespread use of a single, standard language would make for more program portability and for easier use of programs produced at other centers.

It can be argued that such a general language would be too clumsy and inelegant for easy use and that the various features are of such a nature that they cannot all exist simultaneously. There exist languages today which are of sufficient generality that few people are familiar with the workings and use of all of the parts. This by no means makes the languages unusable; the user need only be familiar with those particular features which he is going to use - the others can be essentially nonexistent for him. Supposed incompatibilites of features are often technical in nature and can be overcome with sufficient effort. Other, more basic problems can be sidestepped. For example, the features of high efficiency (of program execution time) and dynamic program modifiability would seem to be irreconciliable. If both high efficiency programs and modifiable programs can exist in the same system, however, both features are obtained to a large extent since, in most applications, very few programs need be kept in the inefficient modifiable state.

This thesis introduces ALAI (A Language for Artificial Intelligence) which possesses many of the desirable features. The details of combining the various features have been worked out and are explained in the relevant sections of Chapter 4. Especially noteworthy is the integration of completely accessible contexts with full backtracking and multiple processes.

3.1 Basic Needs

Users of LISP who are also familiar with a more standard language know the difficulty of expressing in LISP many basic programming operations. For example, the simple arithmetic expression for one solution of a quadratic equation:

```
(-B+SQRT(B*B-4*A*C))/(2*A)
```

must be written in basic LISP as:

```
(QUOTIENT (PLUS (MINUS B) (SQRT (DIFFERENCE (TIMES  B  B)
(TIMES 4 A C)))) (TIMES 2 A))
```

Similarly, the lack, in SNOBOL, of simple iteration statements, 'if' statements, type checking, etc. and the requirement that all operators be spaced off from their operands will often prove irksome to a programmer used to more standard conventions and capabilities. Thus it would seem that standard capabilities are required in a language that is to be used for nonstandard applications. As a result, ALAI has been given the capacity for manipulation of integer, real, boolean, logical, string, array, procedure, etc. values in ways that are familiar to most programmers. For the same reasons, ALAI has been provided with flexible input/output capabilities and the capability for formatting of values to be output and for easy decoding of input strings.

The existence of many standard features in "standard form" tends to require that the entire language be of some "standard form". This essentially means that it is built on top of a standard language of some kind. A large portion of the new languages introduced in the past few years have been based on ALGOL. The reason for this is perhaps the readability and structure of ALGOL, both of which have been praised by computing theorists. To conform to this trend and also to satisfy certain personal preferences, ALAI is an ALGOL based language. This is not to say that all structures and conventions appearing in say, ALGOLW, appear unaltered in ALAI. Rather, the basic form is that of most ALGOL languages, and most features offered in basic ALGOL are available in some form or other in ALAI.

Aside from the standard features of arithmetic, arrays, iteration, etc., several languages offer other data types as basic types. These include sets (unordered, with only one occurrence of a particular item), bags (unordered, any number of occurrences), lists (ordered, any number of occurrences) and tables (one dimensional arrays indexed by arbitrary values). These types are all available in ALAI. Also available are vectors, which are externally equivalent to lists but are represented internally as sequences of consecutive locations rather than as linked structures, and chains, which are bidirectionally linked lists. For completeness, the various standard operators are available, e.g. list concatenation and set union, intersection and difference, along with a few new operators for working with bags.

The iteration capabilities offered by SAIL can be convenient to use. They allow a single iteration to pass over several successive ranges of values. MLISP offers a different version in which several iteration variables simultaneously pass over independent ranges. The total iteration ceases when any of the ranges end. This latter form has been chosen for ALAI. Along with this form is the ability to iterate through the elements of sets, lists, bags, vectors, arrays and tables. Through the use of loc's (pointers to variables, etc.), the iteration facility allows the elements being iterated through to be changed as well as referenced. Thus an array can be initialized without explicitly specifying an index, hence avoiding any necessity of index checking.

3.2 Basic Internal Forms

The previous chapter pointed out the advantages of having an internal program structure which is accessible to user programs. It also pointed out some disadvantages of having a purely LISP-like external form. Some language designers have avoided these difficulties to some extent by building a more standard syntax on top of a LISP-like base. This syntax can be accepted by either an interpreter (written in LISP) which stores the external form internally and executes it as needed, or a translator (again written in LISP) which produces LISP programs equivalent to the input source programs. It is also possible that the entire system be designed with two different representations in mind, one internal and one external. This latter method is used in ALAI, but aspects of the above translation scheme are also incorporated. The only major difference from the translation scheme is that no part of the system is intended for accepting and parsing the internal form. (User programs can of course simulate this with very little effort.)

A facility is available for executing statements during the parsing of the external form of a routine. These parse time statements have access (as do all statements in ALAI) to the internal forms being produced and hence can modify or add to them in any way. In this way a very powerful macro facility, using the full power and convenience of the ALAI language, is provided. It is also possible for the parse time statements in one routine to modify the internal structures of some other routine. This could be desirable for automatic standardization of structures and representation of things used by the programmer. The program which does the parsing (translation from source form to internal form) is of course written in ALAI. It is responsible for syntax checking, type checking, identifier verification, etc. and for most error messages.

The ability to do interactive operations has proved to be of value. One aspect of this is the ability to modify a program to improve its performance. This can be done via text editors working on files containing the source statements for the program or can be done by a component of the interactive system which manipulates the internal structures used for programs. The latter approach is illustrated by the language APL, which provides a form of character oriented editing on the external form of stored programs. The user is thus isolated from the internal representations used. In ALAI, the internal structures are fully accessible to the user, hence the editing can be done directly on them. The editor is a standard program, written in ALAI, which provides facilities for locating and changing parts of internal structures. It works in conjunction with the parser and with a routine for printing internal structures in a form similar to the original source form.

The internal program structures produced by the parser (and any user programs which care to do so) are immediately executable via an interpreter. This method does not produce much efficiency, so a further component of the system is needed. This is a compiler which translates the internal structures into machine code for the local computer. Programs which have been compiled cannot be modified in that state; rather, the original uncompiled version must be changed and then recompiled (or used instead of the old compiled version). Such compiled versions can be freely intermixed with uncompiled ones and are identical in usage except for their greatly increased efficiency. As with the parser and editor, the compiler is simply another ALAI program which is standard for a given implementation.

3.3 Portability and Redundancy

As mentioned above, many parts of the ALAI system will be written in ALAI. This extends to the higher level portions of the interpreter for program structures. Thus the only parts not written in ALAI are the very low level interpretation routines which would directly deal with the interaction between ALAI constructs and the host computer and operating system. In this way, the entire system can be made easily portable. It will also be highly standardized, since there will effectively be only one, standard implementation. A given implementation (except of course the first) need only provide the few bottom level routines, a task which should involve no more than one or two months of effort. Because of the vastly different machine architectures, it would be necessary to transport the ALAI versions of the system in a coded form which represents the linked and sequential structure of the programs in a way which is independent of word lengths and addressing modes.

Ways have been developed whereby the base structures can be portable, the only local portion needed being an I/O interface. An example is the language ALGOL68. The ALGOL68C compiler (written in ALGOL68C) produces machine code for a hypothetical computer called the Z machine. A standard part of the implementation is a FORTRAN (chosen because it exists at most computer installations) program which interprets Z-code. Thus the compiler (the Z-code object for it) is interpreted by the FORTRAN interpreter and produces Z-code programs which are also interpreted by the FORTRAN program. A particular installation which wishes greater efficiency can produce a program which translates Z-code into the local machine code. Although this method makes implementation slightly easier, it will not be used for ALAI, mostly because of a desire to keep things simple (no hypothetical machines or foreign languages are introduced). Also, the local implementer of ALAI can, in the interests of efficiency, hand code any of the routines without fear of violating any conventions (he will have chosen and implemented the bottom level conventions). If greater overall efficiency is needed, a local compiler can be produced, using the initial implementation's compiler as guide and base.

As mentioned in section 2.4, the 2.PAK language contains very little redundancy, i.e. there are usually very few ways of saying the same thing in the language. This has the advantage of making the language easy to learn and describe. At first glance, it would seem that ALAI has taken the almost opposite approach. For example, the 'cond' form is semantically equivalent to 'if' - 'then' - 'elif' - 'fi' sequences and is thus redundant. Its syntax is not the same however, and the external form, not the semantics, is the part with which most users will be concerned. Both the 'cond' and the 'if' are included because they are each preferable in certain cases. For example, 'if A<B then A else B' would usually be considered better than 'cond (A<B:A,A>=B:B)' because of the former's greater readability, but when long sequences of alternatives are involved, many may prefer the shorter 'cond' form. Many programmers have their own specific preferences on language form and would be much happier if their favorite form were used. To keep everyone (at least most of the programmers) happy, both forms are included. Those who do not wish to use say, the 'cond' form, can simply ignore its presence; they are not likely to use it accidentally. Similar reasons hold for several other cases in which some structure is really not necessary.

3.4 The Data Net

Many different forms of data network (net, data structure, association net) have been used by various programming languages. Undoubtedly, most programmers would have definite preferences based on their own particular experience and needs. Unlike programming constructs, the various net styles cannot all be offered without introducing unnecessary inefficiencies into the execution-time storage and computing needs of programs using the net. The best solution would seem to be to offer one general form which can easily and efficiently contain the others. Some data nets (such as that of 2.PAK) use nodes for objects of discourse and labeled edges for properties and relations (concepts). Others represent both objects of discourse and concepts as nodes, and form relationships by pointing to these. The node and edge form as used by 2.PAK cannot easily provide the capabilities of a concept and relationship form. For example, finding all uses of a given concept as a relation may involve searching the entire network for all edges with the appropriate label. No such problem arises with the concept and relationship representation. Another advantage of this second representation is that properties, binary relationships, or relationships over more than two arguments can all be represented with equal facility. Most network forms are representationally equivalent so that, in the end, the decision as to what form to use must be based on matters of convenience and preference.

The data structure chosen for ALAI has two main components: a dictionary and a semantic network. The dictionary provides a means of accessing net elements directly by their name (if any). Any elements which are thus accessible will have direct access to the relevant dictionary entry. The net elements fall into two categories: instances, which correspond roughly to things and propositions; and concepts, which correspond roughly to properties (including states things can be in and activites they can be engaged in) and relations. The significance of nodes in the net is entirely up to the user, but most would correspond to some type of concept or instance; for such nodes certain fields are predefined. Because it is the user who defines the node types, there are arbitrarily many types, and the various types can be used in arbitrary ways. Certain uses, however, are expected to be more common than others. For example, facts or propositions stored in the net would most likely be in the form of compound instances having fields for a concept (the "verb" of the "sentence") and several participants (the "subject" and "object", etc.). When such an instance is created, the ALAI net manipulation routines automatically create access links to it from the concept(s) and instance(s) used in the compound instance. This system of automatic back links provides for completely bi-directional links throughout the network. The back links (usage and participation lists) introduce some storage space wastage, but it is felt that this is a fair price for the ability to traverse structures in either direction.

For the sake of efficiency, an extra dimension is added to the repertoire of possible node types through the use of descriptors. These are identical to instances and concepts except that when the system is constructing back links to compound elements, it will not link from a permanent (non-descriptor) element to a descriptor. In this way, subnets of descriptors can be created which are not linked to and hence can be changed or destroyed without a great deal of backlink modification.

As discussed in Chapter 2, the net structure used by SAIL is designed to be efficient in terms of accessing time. The price for this efficiency is wasteful use of storage space and strict restriction to triples as stored facts. No attempt has been made in ALAI to achieve SAIL type accessing efficiency. Most of the time associated with use of a data net is that of searching through the many links for some specific fact or substructure. The net structure of ALAI attempts to reduce overall search time by guiding the search. Each element in the net has an associated count of how many times it has been used as a part of a compound in the net. A search for an element would then be guided by these counts; the links from little used elements would be followed first, rather than those from elements used often. Efficiency of storage is achieved to some extent by allowing each compound to have as many parts as it needs, thus avoiding many of the linked lists used in LISP representations.

The external form used to represent sentences and substructures to be inserted into or searched for in the net can greatly affect its attractiveness to users. The form used in 2.PAK for producing and traversing node and edge structures is very nice; the sequential nature of the structures is plainly displayed, as are the labels for the edges and nodes. Production of tree-like structures is not quite as easy, however. The use in LISP of bracketed pairs to represent the branches of a node can be expanded to handle n-way branching, but the list notation must be abandoned to avoid ambiguity. Thus a form such as

```
(KNOW (SONOF PETER) (LIKES TOM (DAUGHTEROF PAUL)))
```

can be used to represent a three level subtree representing the knowledge "the son of Peter knows that Tom likes Paul's daughter". The use of indenting to represent depth is also possible:

```
KNOW
   SONOF
      PETER
   LIKES
      TOM
      DAUGHTEROF
         PAUL
```

This method, though representing trees in a readable form, occupies a large amount of space and is dependent on character position, a factor which is usually ignored in ALGOL languages. Numeric level indicators can be used, as can special level shift characters (of which the LISP brackets are one example). The simplest and most concise form would seem to be that used by LISP. This form has been selected for use in ALAI to represent descriptors, sets, bags, lists, chains, etc. The actual type is distinguished by a single character prefixing the structure.

3.5 Pattern Matching

Traversing of structures, done with the same form in 2.PAK as is used to create them, can be done for multi-element instances by specifying the index of the pointer to follow down. Thus 'X:=X(2)' moves the pointer X down into the second substructure of the structure it was previously pointing at. More complicated traversals can be done via the pattern matcher. Note that the LISP method of using CAR, CAAR, CADR, etc. is simply an abbreviated form of the numeric index method, in which A has the value 1 and D has the value 2.

Three different pattern matchers were discussed in Chapter 2. The SAIL net search mechanism and derived sets can be considered to be a pattern matcher. Their operation is restricted to searching the association net. The SNOBOL pattern matcher is powerful, easy to use and concise but is restricted to working with string subjects. The 2.PAK pattern matcher is powerful and general (it can match strings and the data net as well as any other structure or data type used) but is very verbose and does not provide many features to make the programmer's task easy. It would be very desirable to combine the good points of all three, along with those from other pattern matchers, such as those used by PLANNER, CONNIVER, QA4, etc. Can this be done?

The difficulties are those of avoiding ambiguity while maintaining freedom of form and conciseness: For example, does

(2.64 "cat" "dog" (12+16))

represent a compound with relation 2.64 (??) and participants "cat", "dog" and the compound (12+16) or what? The "cat" "dog" pair can be a sequence of elements of the main structure being defined or can be a separate subpattern representing the single string "catdog". Ambiguity comes from the unspecified nature of the sequences being represented. This can be overcome by fully specifying the type of such sequences and by requiring that all subsequences be properly typed. The already existing convention of single character typing prefixes for sets, lists, etc. can be carried over and extended to include other types such as strings and bit sequences in pattern specifications. In ALAI, a further convention is introduced which states that bracketed elements within a sequence are subsequences of the same type. Thus the above pattern can be written unambiguously as say,

L(2.64 "cat" "dog" +(12+16))

which represents a list containing the real number 2.64, the string "cat", the string "dog" and the result of evaluating the expression +(12+16) or as

S(2.64 string("cat" "dog") (12+16))

which represents a set containing the real number 2.64, the string represented by the pattern "cat" "dog" and a set containing the result of evaluating the expression 12+16.

The SNOBOL convention of a blank for concatenation and

'|' for alternation has been used. Also added are several further conventions and symbols, most of which generalize the SNOBOL pattern functions and values. Even more conciseness (and perhaps a little confusion) has been achieved by letting symbols which have one meaning in normal expressions have another meaning in patterns (a break character is provided to switch back to normal evaluation). For example

```
   string(("dog"@4 "cat")=S1)
```

matches the string "dogdogdogdogcat" and assigns the portion of the subject so matched to the variable S1. Special forms are also introduced for representing integers, real numbers and boolean values. The result of these choices and conventions is a pattern matcher which is general like that of 2.PAK but which is approximately as concise (more in some places, less in others) as those of SNOBOL and SAIL.

The syntactic form of the net search mechanism is also significant. Many net searchers are of an "all or one" nature in which there is little control over what is retrieved after the initial specification has been given. It is often desirable to limit the number of samples found or to eliminate some for some reason or to perform some other associated operation in between searches for successive samples. The iterative construct, especially in the generalized form offered by ALAI, allows these manipulations quite readily. Thus the net search construct in ALAI is an expression form usable in the 'via' form (section 4.22) of the iteration statement. This method has the advantage of allowing greater control of and intermixing with net searching without introducing much in the way of new language constructs.

3.6 Backtracking

A feature of artificial intelligence programming languages which has received much attention lately is that of backtracking. The amount of backtracking offered by a given language can vary from very general forms which are easy to use and are a vital part of the language to simple forms involving nothing more than some context mechanism which can store variable bindings. The full power of the technique, which can be thought of as an implementation of the general heuristic "trial and error", was perhaps first realized only after the appearance of the PLANNER language. Since then it has been incorporated into several languages in several different forms. The technique, though very powerful and general, is extremely inefficient in terms of computation and as such has recently encountered increasing opposition. It is argued that backtracking does not represent human problem solving methods accurately, nor does it easily allow any heuristics to be used to decrease the search time. Planning is not involved, and thus the problem is not solved "intelligently"; an acceptable solution is merely found by brute force of computation. It is also argued that most problems which are solved by "automatic failure driven backtracking" techniques can be solved much more efficiently by conventional techniques if a little thought is given to the problem and how to solve it.

It cannot be denied, however, that the trial and error method can produce solutions to problems which are not amenable to more sophisticated analysis. The question becomes that of which problems cannot readily be solved by other means and if in fact there are enough of them to merit the effort needed to incorporate backtracking into a language. As with most supposedly decisive questions, the answer will vary from person to person. One who is interested in the theoretical aspects of problem solving and is willing to spend the time and effort needed to analyze problems to an extent where purely algorithmic solutions are possible is likely to consider failure driven backtracking unnecessary and in fact an undesirable evil. On the other hand, one who is interested only in solving some fairly straightforward problem with minimum effort on his own part, is likely to consider such backtracking as a definite boon.

Can a middle ground be found, i.e. is there a solution acceptable to both extremes? Obviously, the technique cannot be abandoned altogether. Rather, it must be made more flexible, more efficient and more theoretically "pleasing". Basically, what is needed is a means whereby the process can be easily controlled and guided without losing any of its inherent power. Consider for example the problem of the missionaries and cannibals (Appendix D). Failure driven backtracking can yield a solution to this problem as long as it doesn't get into a loop of some kind. In doing so however, it will waste large amounts of time trying possibilities which are equivalent to others already tried or that can be foredoomed to failure with a minimum of computation and thought by the programmer. It is not, however, at all obvious how to solve the problem in a purely deductive way, i.e. to produce some algorithm which produces solutions for arbitrary starting situations. Guiding the solution via heuristics, timely checks and the use of a reasonable problem representation can improve things immensely. For example, a little thought will show that much effort is saved if the boat takes as many people as possible on the trips over and as few as possible on the trips back. Hence a way is needed whereby the order of choices can be controlled. It is also pointless to try taking more cannibals than missionaries (unless no missionaries are taken), so that early failure or simply not trying some alternatives is desirable. To enable early detection of doomed possibilities in such a way as to minimize computation, it is necessary to test the prospective values before asserting them in the context to be used for backtracking. Thus one would like to be able to chose from a pretested set of values. These requirements are similar to those for net searching, hence in ALAI, they have been handled similarly. The backtracking form used in ALAI is very similar to the basic 'for' iteration statement and has its full capabilites for iteration through sets, 'while', 'until' and 'suchthat' forms and the use of parallel iteration. This format allows the easy combination of net search with backtracking, so that alternatives can be supplied from a semantic base.

The "goodness" of a given try can change dynamically as more is learned about it and the other trys, hence it is desirable to be able to discontinue one line of investigation in favour of another. This situation can be nicely handled by using processes for the trying of the various alternatives. They can then be run in parallel or sequenced by a knowledgeable scheduler so as to take advantage of their expected usefulness. They can communicate with each other and with the scheduler by means of events and global switches. In ALAI, a second type of "backtracking" construct, again with the basic form of the iteration statement, is used to produce the processes with the various values tried being produced by the iteration forms.

Intimately associated with the backtracking facilities of a language are the contexts used to store alternate values. In ALAI, the form required is one with automatic provisions for saving values which define the various alternatives. (A less powerful or more user controllable form would be redundant with the control given by the iteration forms and would be more difficult to use.) To ease things further, any part or subpart of a variable can be made automatically saveable in all contexts. The contexts themselves are completely manipulable and have been designed to fit in with the idea of multiple processes. Those who wish a simple storage bin type of context can use tables indexed by the location to be saved.

As indicated above, a form of process exists in ALAI. Of the languages discussed in Chapter 2, SAIL and 2.PAK had the capability for forms which could be called processes. Other languages not discussed, such as QA4, have similar capabilities. The type of process facility actually required will of course depend on the particular user and application. Once again, the choice for ALAI, which trys to be useful for everyone, is to offer a full power facility providing all of the capabilities that may be needed. The result is similar in nature to the facilities offered by SAIL and QA4, but is different in syntax. Each process must be able to run independently of the others in terms both of variable and net values and of when and how long it is running. It is desirable to have a means of interprocess communication by which the processes can affect one another and can synchronize themselves. As with SAIL, this communication is effected with events and event types. An event is something that can have a very definite semantic meaning, hence it will often be necessary to represent events in the data net. To avoid duplication of effort and to provide a meaningful link between the abstract quantities in the data net and the hightly structured, highly definite programs, the events used in ALAI are valid elements of compounds and descriptors. They can be retrieved through data net search, and data net semantic knowledge concerning them can in turn be retrieved directly from the event itself.

The actual contents of processes and events and the creation and destruction of them is controlled by the user. It is necessary, however, that events and processes, possibly running in parallel, fit in consistently with the pre-existing notions of backtracking and contexts. For example, what is to happen when a process which has created and started other processes which are still in existence and can be resumed, fails back past the points at which the other processes were created? If the subprocesses are left active, backup past their creation point would change and perhaps destroy (by backing up past a block entry point) values which the subprocesses are dependent on. The backup can be stopped at the process creation points, or the processes (and all processes they created, etc.) can be destroyed. Similar cases hold for a backup of one of the created processes. Because the action taken is a major one and can greatly affect the program(s) currently running, ALAI gives the choice of action to the user. The choice is specified in the form of the relation between the creating process and the created process at the time of process creation.

One of the arguments used against backtracking is the fact that contexts provided for use by the backtracking mechanism are very inefficient in terms of computation used to switch between them. The amount of computation required can become very high in a language like ALAI where complete freedom with contexts is given; the user can switch between any contexts at will and can reference values in other contexts. If the user wishes to work in environments which are highly dissimilar, e.g. with respect to the data net contents, then switching between contexts representing the environments can be a very lengthy process. The problem can be avoided by preventing such use of contexts (which perhaps explains the lack of power of "contexts" in some languages) but this solution is not very appealing for a language which is trying to be general. In ALAI, two methods are available for speeding up environment changes; one involves making relevent contexts "closer together" and the other involves a new data type, "world". A world is a complete environment, including a full data net and dictionary. Switching between worlds ("entering" the new world) involves changing only one or two pointers indicating which is the current world to use for assertions and net searches.

3.7 Planning

One of the intended uses of the ALAI language is as a means of specifying the process by which an "intelligent" system can produce a plan or program which accomplishes some desired goal. In its early stages of formulation, such a plan would be represented as a subnet of internal data structures. This type of representation would have to be interpreted by the user's program and hence would not be very efficient. The final form of the plan would be a valid program structure which could be compiled into fast machine code. The step between the two forms is a very large one and would likely prove a substantial obstacle to the plan making process. What is needed is some intermediate form, perhaps allowing an entire range of representations. One way of doing this is to allow a mixture of program structure and semantic structure which can be executed by the system. This method is done by ALAI "elaborations". They are directly executable as parts of programs, but can contain arbitrary semantic structures which are to be elaborated (accomplished) by the current elaborator routine. This form is something of an experiment in ALAI; they have not yet been used as ALAI has not yet been implemented, so that their true usefulness is not known. A more detailed description of their nature and how to use them can be found at the end of the next chapter.

CHAPTER 4

THE LANGUAGE

4.0 Method of Description

Language descriptions for ALGOL type programming languages are often based on a complex B.N.F. grammar. The grammar given for ALAI (Appendix A) describes the language syntactically, but does not incorporate all of the features scanned for and checked by the parser. The relevant parts of the grammar are meant to be read in parallel with the specific parts of the language description. The descriptions in this chapter provide examples of the various constructs and provide semantics for them, i.e. tell what they mean. Before the more detailed descriptions are given, a short, generalized introduction to ALAI will be given, in order to give the reader a grounding in its basic structure.

4.1 Basic Structure

ALAI is an ALGOL type block structured language, but it has no 'begin's and its 'end's are used to match 'attempt's. Instead, existing "bracket words" have been extended to act as block delimiters, and more control words have been added. By using special control words that fit in standard configurations, it is easier to find what part of a program is a unit, i.e. where it starts and where it ends. For example, 'fi' always matches an 'if', 'od' matches a 'do', 'esac' matches a 'case' etc., rather than having 'begin' 'end' pairs for all blocking structures. If a block is needed (i.e. as a sequence of statements where only one is allowed) then the delimiters 'do' and 'od' or brackets can be used. An example is the 'failing' statement:

```
failing do A:=B; C:=D; MERGE(X,Y); fail od;
```

Note that in this description (and hopefully in user programs), special words are written in lower case, while user declared identifiers are written in upper case. Language defined identifiers other than keywords will often begin with an asterisk, which is not allowed in user declared identifiers.

When used interactively, ALAI will accept statements for immediate execution, hence there can exist no explicit "main program"; instead, procedures (procs) represent top level programs. Procedures can be recursive and, as with other blocks, program execution can back into them even after they have been exited. (This is only possible when there is a backtrack point somewhere in their range.) The ALAI parser is meant to be used in a directly interactive situation where errors in source language input are detected and flagged immediately after the current line is entered. The effect of the parser is to produce internal structures which can be interpreted (executed) by the interpreter and can be compiled into machine code by a compiler. As such, the parser must effectively operate in 'one pass'. To do this, it is necessary that all variables, record classes, procedures, etc. be declared before they are used. This restriction is removed for label and elaboration constants so that it is possible to branch forward and to instantiate an elaboration before it is encountered. Procedures and record classes which reference each other are handled by the use of dummy declarations which inform the parser of the identifiers to come later without actually defining those identifiers.

When ALAI is used interactively, top level statements (i.e. not inside a procedure declaration) are executed immediately. In this case, and in this case only, expressions are valid statements; they are executed and their value is printed. Such statements and expressions are executed in an environment containing declarations which are global to all user-defined procedures. Variables, record classes, data types, etc. which are of this type are declared whenever needed (as long as the user is in the top level of execution and the declarations come before all uses of the things being declared) and are available to all procedures defined later. The objects thus declared are called 'fixed' objects, e.g. fixed record classes, fixed variables, etc. Statements and expressions entered at this top level are not executed until they have been completely parsed. Procedure calls in the top level are used to start the execution of the procedures which are effectively the main programs. If ALAI is not being used interactively, then the top level statements and declarations represent the single 'main program', just as in most other ALGOL type compilers.

Conditional constructs in ALAI are 'if' statements (end with 'fi'), 'if' expressions (do not need a closing 'fi'), 'cond' statements and expressions, 'attempt' statements and expressions, two types of 'case' statements and expressions, parallel iteration ('do', 'try' and 'ptry') statements, 'for' expressions and 'failing' statements. Their meaning is usually discernable, but in case of doubt, consult the relevant section of this description for full explanations of syntax and semantics. The occurrence of the pseudofunctions 'S', 'B', 'V', 'P', 'T', 'L', 'C', or 'D' causes parsing of sets, bags, vectors, pairs, triples, lists, chains or descriptors, respectively, in which all bracketed sequences are considered to be of the same specified type. The special brackets '<' and '>' instead of normal round ones prevent the type from being applied to the inner elements (they must supply their own type). For example

```
L(2 3 (6 1 A+B) (641+1))
```

produces a list containing two integers and two lists, whereas

```
L<2 3 (641+1) <A Q>>
```

produces a list containing three integers and a set. The special brackets without any preceding pseudofunction yield a set. When parsing descriptions (pseudofunction 'D'), which includes patterns, the occurrence of a data type (e.g. integer, string, list) causes parsing of a pattern of that type. Many types have special pattern constructs, but the basic operations of and ('&'), or ('|') and iteration ('@') are usually available.

The constuct 'A@B' is used to reference field B of record A. To get a 'loc' (pointer to the actual storage area reserved for a variable or element) or 'procval' (pointer to the internal structure representing a procedure) value, the needed variable or proc name is preceded with '@'. To de-reference such 'loc's and 'procval's, the '&' prefix is used. i.e. '&@X' is equivalent to 'X' for X either a program variable or a procedure. '@' also returns specific fields of specific records (values of type 'loc') as in '@A@B'. Unevaluated expressions are created by the '#' prefix and evaluated by the '=' prefix. '+' before a statement yields parse time evaluation (a macro). An input line with a "C" in the first column is treated as a comment, as are all characters enclosed by the pair "/*", "*/".

4.2 Identifiers

An identifier in ALAI is a sequence of one or more letters, digits and underscores which starts with a letter. There is no upper limit on the size of identifiers. Some language defined identifiers are prefixed with an asterisk

(*). Identifiers are used to denote variables, fields of records, procedures, data types and data type elements. Several 'reserved words' i.e. identifiers that have a special syntactic meaning, exist in the language. They cannot be used as identifiers since the scanning routines pass them to the parser as special reserved words and not as identifiers. The parser accepts both normal identifiers and reserved words in either upper or lower case so it is advisable to be aware of all of the reserved words. The reserved words in ALAI are:

active, add, all, any, array, as, assert, attempt, by, card, case, cause, collect, cond, corp, delete, div, dnoc, do, dummy, elif, else, empty, end, enter, esac, exit, failing, false, fi, find, find1, for, fresume, from, go, goto, if, in, inactive, inbuff, incarnate, is, isnt, iterate, line, need, next, null, od, of, on, outbuff, pop, priority, proc, ptry, push, rem, reset, resume, return, sign, suchthat, then, time, to, true, try, until, using, via, wait, while, with, wrt.

4.3 Declarations

Declarations inform the parser of the existence of a valid identifier and of the way in which the programmer intends to use it. All identifiers (except those whose pre-declaration is part of the language) must be declared before they are used. No identifier may be declared more than once in any block, unless all such declarations are as field names for different record classes. A declaration of an identifier in an inner block will override, for the duration of that inner block, any declarations of the same identifier in an outer block. (Pre-declarations by the language are considered to be in a block which contains all of the programmer's fixed procedures and variables.) An identifier is defined (i.e. valid) only in the block in which its declaration appears. Outside of that block, references to the identifier will be in error. All declarations must appear before all executable statements in a block.

As is seen in the grammar in Appendix A, ALAI declarations allow initializations, i.e. the assigning of an initial value to a variable being declared. In ALAI, when an initialization is parsed, the parser removes the initialization and inserts an assignment statement before the executable portion of the block to do the initialization. The assignment statements are in the same order as the initializations which caused their insertion. The most common and straightforward type of declaration is the <normal-declaration>. Declarations of this type correspond to standard declarations in most programming languages:

```
int A,B,APPLE_OF_MY_EYE
event list stack back static COLLISIONS
bool static FLAG:=true,DONE:=false
back X,Y
ref(PERSON) PRESIDENT,CHAIRMAN
loc stack list loc P, Q
```

The type 'free' (which is the default if no <simple-type> is specified) allows values of any ALAI data type, including such things as integers, bits, expressions, procedures, arrays, net references, processes, etc. Such values have the type associated with the specific value and this type can be found and checked by an executing program.

Normal vectors, strings and arrays are of varying size and are represented as a pointer to a special record which defines the vector, string or array. If the forms 'string (<number>)', 'vector (<number>)' and 'array (<boundpairs-1>)' are used, however, the strings, vectors and arrays so declared are of fixed size and are stored directly in the data area for the block, rather than in some dynamically obtained area which is pointed to only by a single pointer in the data area. References to them as strings, vectors and arrays yield copies which are of the standard record and pointer form. Assignments to them will cause copying of the values into the fixed areas. If the value thus assigned is too large, truncating of the higher positions occurs; if the value is too small, padding of the higher positions with blanks or nulls occurs. It is meaningful to declare such variables as 'backall' but not as 'back'.

Sample fixed string, vector and array declarations:

```
string(300) SENTENCE:=" "
int vector (4) local QUAD,FOUR:=V<3,4,9,-127>,GROUP
array(6,-12:32)backall global MATRIX,ELEMENTS
```

If the form 'array (<dimensions>)' is used, then the array is represented as in varying sized arrays, but the parser will check that the number of dimensions (indices) is always correct. E.g.

```
string list array (*,*,*,*) P
```

declares P to be a four-dimensional array, but its size is not specified. Note that forms such as:

```
string(12) array (15:27)
```

though unambiguous and perhaps desirable, are not allowed. Ambitious implementers may allow and support such features, but the standard implementation will not.

Record declarations merely specify the content of the declared record class. Such declarations define only the form of the records, hence it is not meaningful to initialize them. Special modifiers are applied to the reference variables of the class, rather than to the class itself. Sample <record-declarations>:

```
record DATA(int I,J; ref POINTER;context MYWORLD)
record TREE(ref(TREE) list CHILDREN; free VALUE)
```

The type 'ref' without the bracketed record class, allows values to be pointers to any user defined record class. A record class with an empty field list can be re-declared in the same block. This allows the creation of record classes which reference each other. e.g.:

```
record A;
record B(ref(A)ptr; free X,Y);
record A(ref(B)ptr; int P; string Q)
```

Declarations of new data types (<user-type-declaration>) specify the name of the new data types and list the identifiers which are to represent the permissible values of those data types. Such new types are equivalent to standard ALAI data types in that they can be used as the type for variables, fields of records and compound types, procedures, etc. The simple type 'any' allows values to be of any user defined type (not records). Example:

```
    type PARENT (MOTHER,FATHER);
    type DAY (MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,
       SATURDAY,SUNDAY);
    PARENT X,Y;
    DAY array (10) local static P;
    record DATE(int YEAR,TIME; DAY DAY1)
```

Procedures can have any number of parameters, including zero, hence forward referencing cannot be accomplished as with record classes. Instead, <dummy-specification>'s are used in which the word 'dummy' replaces the word 'proc' and only the procedure name, its type (if any) and the names and types of its parameters (if any) are given. Note that an untyped or proper procedure does not have type 'free' or type 'null' or any other valid ALAI type. Proper procedures cannot appear as parts of simple expressions as they return no value. Similarly, typed procedures cannot appear as statements. Procedures with a variable number of arguments are handled via the <multiple> syntax. Only one multiple parameter is allowed, and it must be the last one. It is denoted by pluralizing the last word of the type. Inside the body of the procedure, multiple parameters will occur as single stacks of the specified type with the supplied arguments pushed onto them last to first (so that the top element is the first one given, etc.).

Sample procedure declarations and specifications:

```
int dummy A(int X);
int proc B(int X,Y);
   (A(X)+A(Y))/A(X+Y);
int proc A(int X);
   if X<0 then 0 elif |X>4 then 10 else B(X,X-4);
proc P(string array PA; int I,J; process list OPTIONS);
   resume OPTIONS(I) with PA(J);
   fail
corp;
```

The following procedure finds the mean of an arbitrary number of reals:

```
real proc MEAN(event_type ERROR; reals X);
   (real SUM:=0.;
   int COUNT:=0;
   while ¬empty X for COUNT from 1 by 1
      do
      SUM:=SUM+<X;
      pop X
      od;
   if (SUM:=SUM/float COUNT)<.5 then cause ERROR fi;
   SUM);
...
Q:=MEAN(BAD,1.,6.,2.75,-37.,16./float J+12.0)
```

MEAN will cause a real division by zero error if it is called with no arguments other than the event type, but the parser will allow such usage.

The <procedure-equivalence> construct gives one procedure the same body as a previous one (and hence the same parameters and type). The special parameter 'string *CALLED' does not correspond to an element of the argument list when the procedure is called. Instead, it is automatically given as value a string containing the name by which the procedure was called. This facility is useful for natural language work where words are represented by procedures. A common procedure, say NOUN, can process most simple nouns; all it needs to know is the name it was called by (i.e. the word it is to process).

Equates (<equ-declaration>) are parse-time free variables which are local to their block of declaration. Uses of them, both referencing and assigning, can be freely intermixed with normal identifiers, but the actions will be done at parse time. For example:

```
free X; bool Y; set S;
equ A,B:=3,C:="This is a message.";
X:=if Y then B+A:=2 else C;
S:=<A,C,"so is this">
```

is equivalent to :

```
free X; bool Y; set S;
X:=if Y then 3+2 else "This is a message.";
S:=<2,"This is a message.","so is this">
```

If the current value of an equ is an expression (is of type 'expr'), then that expression is inserted into the code, rather than appearing as an expression constant.

ALAI is a block structured, recursive language. For ALAI, this means that when a program is executing, entry into a block with local variables necessitates the acquiring of storage for those variables. Such run-time storage allocation is necessary to insure that separate calls to a procedure which uses itself recursively all have separate storage areas. Separate storage for each variable on each invocation is not always desirable, however. Procedures which are not recursive can be speeded up if all of their local variables (including fixed size strings, vectors and arrays) and parameters do not need to be provided storage each time the procedure is called. A programmer may also desire that a procedure remember some value between calls, but may not want to go to the trouble of making the procedure into a process and using it as a coroutine. To allow these types of things, ALAI has the special modifier 'static' which can be applied to local variables and to parameters. Such static variables have only one storage area set aside for them (other than when saved in a context) and these locations are used whenever the variables are referenced, regardless of any recursion or multiprocessing.

Another aspect of block structured programming is that local variables and anything referring to them lose all significance when the block of their declaration is exited. The parser must prevent all attempts to pass a possibly local value out of its range of declaration. Some things, however, are global objects, e.g. an array of integers, a list of references to a fixed (declared external to all procedures) record class, etc. even though they are currently the value of a local variable. Since the parser cannot know whether, say, a stack is local or global (the values are put onto the stack at run time, hence the parser cannot be sure of their type) it would have to prevent all attempts to assign possibly local values to any variable which is more global than the current block. This is a rather large restriction.

In ALAI, this problem has been partly overcome by allowing two types for such objects of indeterminate globality, declared by means of the special modifiers 'local' and 'global' ('local' is the default). 'Global' variables can receive only values which are guaranteed to be totally global. Their values can then be transmitted out of their block of declaration, can be attached to the data net, can be passed to other processes, etc. 'Local' variables can receive any value (of the proper ALAI type), regardless of its scope, but these values cannot be passed out of the range of the variable. Constants which are local, e.g. pointers to local variables or labels, can only be assigned to 'local' variables whose range is no greater than that of the constant. It is felt that this arrangement allows the programmer to do all value transfers which are guaranteed to be meaningful. If, for some special purpose, it is desirable to circumvent the 'local'/'global' arrangement, the programmer must do so via more devious means (e.g. going through system constructs to assign local values to global variables).

Many AI languages which offer backtracking do not provide much control over the nature of the backtracking. In ALAI, global switches control backtracking (i.e. saving of changed values in the current context) separately for the data net and for program variables (fixed variables are considered to be program variables). Individual control over each variable is allowed (when the global save switch for variables is off) through the special modifiers 'back', 'backall', 'backallall', etc. Any assignment to a 'back' variable or to a top level element of a 'backall' variable will be saved in the current context. 'Back' is relevent for all data types except fixed size strings, vectors and arrays. These types can be made backtrackable by using the variable sized versions. 'Backall' is relevant only to all compound types and to 'free' and 'ref' variables. Compound types which are declared 'backall' have assignments to all of their elements backtrackable, just as if each element had been declared 'back'. Note that 'backall' is a property of the variable and not of the particular object (e.g. array) which is the current value of the variable. It is thus meaningful to declare fixed size vectors and arrays as 'backall'. 'Backallall' bears the same relation to 'backall' as 'backall' does to 'back', etc. Note also that 'backall' does not automatically imply 'back'. For example, if L is an 'int list array', i.e. an array of lists of integers, then if it is 'back', then assignments to L itself are saved; if it is 'backall', then assignments to the elements of the array L are saved and if it is 'backallall', then assignments to elements of the lists in the array (i.e. the integers) are saved.

4.4 Expressions

Expressions are built up of constants, storage location references and operators. Constants are of varying types and varying forms. Storage location references are most often just simple variables but can be much more complex, such as

```
/ref (& /procval &(M(12,I+J)(16)))@(J div 2)
```

which can mean "the J-div-2nd field of a record which is returned as the value of a parameterless procedure which is the current value of a location pointed to by the 'loc' at the 16'th position in the vector which is the current value of the 12,I+J'th element of array M". (In order for the parser to accept this location, M would have to be declared as 'loc vector array' or something even more specific.) Operators are usually simple language defined ones such as '+', '-', '&', '¬=', etc. but can also be the names of user defined procedures.

The structure of the various location forms can be seen by referring to Appendix A. If a location is the left side of an assignment, then the assignment puts a value into that location and returns that same value as the value of the assignment. Other references to locations refer to the current values in the locations. A simple identifier (some language defined identifiers are prefixed by a '*', but are still just identifiers) refers directly to the location associated with that identifier. The location which is the value of a 'loc' expression can be referenced by prefixing the 'loc' expression by the indirection operator '&'. For example

```
&(if B then @X else @Y):=3
```

assigns 3 to either X or Y, depending on the value of the boolean, B.

The '@' operator when used as an infix, indicates field selection in records. The record being selected from is the result of any expression of type 'ref' or whose type is that of a pointer to a single user defined record class. The selector, although it can be any integer expression, is usually just the identifier associated with the required field of the record when the record class was declared. Array subscripting is standard except that the array being subscripted can be the result of a complex expression or procedure call. Strings can be referenced in their entirety or through a substring designator. The substring designator indicates the starting position of the substring within the given string (the first position is position 1) and the length of the substring. (If not specified, the length defaults to 1.) For example, 'S(2|34)' refers to the 2nd through 36th characters in the string S. Bits values can have subsequence designators which are entirely equivalent in function to substring designators. Such substrings and subsequences can be both referenced as expressions and assigned to.

Vectors, lists, chains, pairs, triples, stacks and queues are all ordered sequences of values and as such can be indexed by a single integer. The indexing starts at the left with index 1 and must be within the size of the object being indexed. For stacks and queues, indexing starts at the top and front respectively. Tables are essentially one-dimensional arrays which can be indexed by any type of

R

value. Indexing with a value which is not yet in the table will create the entry if the table reference is the left hand side of an assignment, and will return null otherwise (but will not create the table entry). For example, if the table T does not contain an entry indexed by "cat" then referencing 'T("cat")' will return null but not affect T, but executing 'T("cat"):="meow"' will create an entry in T, indexed by "cat" and give it the value "meow".

A subexpression which is enclosed in brackets is clearly delimited, i.e. it is clear where it starts and stops, hence such an expression can be arbitrarily complex without introducing ambiguity concerning its bounds. To take advantage of this clarity, ALAI allows block expressions to be used wherever bracketed expressions are used. Block expressions are zero or more declarations followed by zero or more statements followed by an expression which is evaluated as part of the block and whose result is the value of the block expression. The whole thing can be enclosed in a pair of identifiers (they must be the same identifier) which serve to comment the limits of the block expression and to provide a name by which the block can be referenced. A sample block expression, named "example", which returns an integer result is as follows:

```
("example" int A,B:=0;
for A from 1 to 6
do B:=B+A
od;
B "example")
```

For procedures which have only one parameter, the brackets enclosing the argument on a call to the procedure can be omitted. The procedure name is then essentially a prefix operator with very low precedence (the argument is expanded as far as possible, up to the next enclosing bracket or use of a procedure name as an infix or prefix operator). In a similar manner, two-parameter procedures can be used as infix operators.

Each expression has an associated type, which is obtained from the location or procedure call which constitutes the expression, or from the last operation performed in evaluating the expression. Procedure calls, assignments to typed locations and operators often require expressions of specific types. When the type of an expression is determinable, it must be acceptable in the position in which it is to be used. Often, however, the type of an expression is not known until the program is actually run, thus the parser has no way of determining it. This can occur when referencing locations which are 'free', such as 'free' variables, record fields selected by expressions, elements of untyped arrays, etc. .

To be sure of correct action, the parser should generate program structures which will check the type of such expressions before using them in places which require specific types (e.g. integer addition). Such "run-time type checking" is time consuming, however, and thus not always desirable. Often, the programmer knows that a particular expression is of a particular type (he might have just checked it) and does not wish run-time type checking to occur. To accomodate this and to reduce the number of type mismatch errors which are essentially syntactic in nature, ALAI requires that all expressions either be given a type by the programmer, have an implicit type, or have the default type 'free' which cannot be used in any position which does not accept 'free' values. The programmer can give an expression a type by prefixing it with either a single or a double slash ('/') followed by the type. A single slash causes the parser to generate program structure which will check to make sure that the expression is of the specified type, and a double slash guarantees to the parser that the expression will be of the specified type and need not be checked. When using the second form, the programmer can of course lie about the type, but any problems arising from doing so are then his own fault. The parser will not accept incorrect type specifications for expresssions for which it knows the type.

The type 'free' encompasses all other types, i.e. locations of type 'free' can have any value whatever. Type 'any' encompasses all user defined data types; type 'ref' encompasses all user defined record classes; type 'fixed loc' encompasses all 'loc's which point to variables whose values are stored directly in single words, i.e. are simple integers, reals, bits values or boolean values; type 'ptr loc' encompasses all other 'loc' values, e.g. 'free', 'array', 'event', etc.

As previously mentioned, the value of an assignment is the value being assigned to the receiving location. Thus assignments are valid expressions. When assignments are made, the expression being assigned must be of the same type as or encompassed in the type of the location being assigned to. Certain assignments, such as to fixed length strings, may modify the value before doing the actual assignment.

The standard form for procedure calls is that of the name of the procedure followed by a bracketed list of its arguments. If the procedure has no parameters, then the bracketed list can be empty or can be omitted. The standard form must be used if the procedure has more than two parameters or if the procedure is not named directly but is the value of a procval expression (prefixed by the '&' de-referencer).

Many other expression forms (e.g. 'A+B' for addition) are allowed by the language. These will be discussed under the relevant data type or in separate sections devoted to special constructs (like 'if' statements and expressions). Expressions are evaluated in a left-to-right manner unless some other order is imposed by the use of bracketed subexpressions. An exception is the case of assignments; the assignment is made before the value of the assignment is returned. Data types for which many operators are defined (e.g. integers) will define a precedence ordering among the operators. Also, if an operator returns a result of a different type than its arguments, then the expressions representing the arguments must be evaluated fully before the operator can be applied. An example of this is the expression in the following procedure, which has only one valid order of evaluation:

```
   bool proc IFBIT(bits X; int POS);
      ¢1 shl POS-1 & X¬=¢0;
```

Such a procedure is not needed in ALAI since single bits are valid boolean expressions, but its form serves well to indicate how typing needs can disambiguate expressions. (The ¢'s delimit bits constants, 'shl' requires a bits and an integer argument and returns a bits result, and '-' requires and returns integer values.) The same expression, fully bracketed is as follows:

```
   ((¢1 shl (POS-1))&X)¬=¢0
```

4.5 Statements

A statement in ALAI can be an assignment (exactly like assignment expressions, except that the returned value is ignored), a procedure call (again exactly like expression procedure calls, except that the procedure must not return a value, else either a parse time or run time error will occur), the empty sequence (this is a convenience only and has no semantic significance), a special statement form (discussed later in relevant sections), or a block delimited by 'do' and 'od'. Statement blocks are equivalent to blocks (normally delimited by 'begin' and 'end') in other ALGOL type languages. They are actually a special case of the general 'do' statement and are included here only to indicate their use as standard blocks. As with block expressions, the identifiers just inside the block delimiters are a construct for naming the block and for marking its ends more clearly. If a block contains no declarations, i.e. no local variables, then it is semantically just a sequence of statements (possibly with a name) in that entry into and exit from it require no storage or backtrack point manipulation. When a block is exited, all labels and variables defined or declared within that block become undefined or undeclared. They can become defined again by re-entry of the block, either from its top through normal program execution, or from its bottom by a backup to a backtrack point established within the block. On such a backup, all local variables will be given the values they had when the backtrack point was established. Note that a resume to another process does not cause the exiting of the block containing the resume. Each instance of a block (several may exist simultaneously due to recursion or multiple processes) has its own copy of all local variables not declared static. Static variables are common to all instances of the block.

Sample statements:

```
SORT(X,Y,12)
P:=Q-12

do int X;
```

```
   X:=3/Y;
   do THING
      real PI; process PR;
      PI:=3.14159;
      X:=X/trunc PI
   THING od
od
```

4.6 Arithmetic

Standard ALAI offers two types of arithmetic, integer ('int') and real. Particular versions of the language may offer other types, e.g. 'short int' or 'long real', but these are non-standard and should not be used in programs which are intended to be written in 'standard ALAI'. The basic form of integer and real expressions can be seen in the grammar. The infix operators '+', '-', '*', '**', '/' and 'mod' represent the operations addition, subtraction, multiplication, exponentiation, division and remainder respectively. The prefix operators '-', '|' and 'sign' represent negation, absolute value and the signum function respectively. 'Sign' returns an integer indicating the sign of its argument: +1 if positive, -1 if negative, 0 if 0. There is no implicit conversion between the two types, i.e. expressions such as '2*3.27' are illegal. Conversions must be done explicitly via the procedures float, trunc and round.

Arithmetic expressions are evaluated in the standard left-to-right order but a precedence scheme is superimposed. Operations with higher precedence are performed before those with lower precedence. The precedence grouping, in order of decreasing precedence, of the ALAI arithmetic operators is as follows

```
-(prefix), |, sign
**
*, /, mod
+, -(infix)
```

Arithmetic expressions are represented internally as calls to procedures which do the required operations. This convention, also used for most other basic operators, allows an internal format which users can easily manipulate.

The arithmetic procedures predeclared in the language are:

iplus - integer sum of two integer arguments

itimes - integer product of two integer arguments

idiff - integer result of first integer argument minus the second

iipower - integer result of first integer argument to the power of the second

idiv - integer quotient of first integer argument divided by the second

imod - integer remainder of first integer argument divided by the second

iabs - integer magnitude of integer argument

isign - integer result of integer argument: +1 if positive, -1 if negative, 0 if 0

ineg - integer result is negative of integer argument

irand - random integer selected from the range of 1 to the argument passed to 'irand'

rplus - real sum of two real arguments

rtimes - real product of two real arguments

rdiff - real result of first real argument minus the second rdiv - real quotient of first real argument divided by the second

rmod - remainder of first real argument divided by the second

rrpower - real result of first real argument to the power of the second

ripower - real result of first (real) argument to the power of the second (integer) argument

rabs - real magnitude of real argument

rsign - integer result of real argument: +1 if positive, -1 if negative, 0 if 0.0

rneg - real result is negative of real argument

urand - uniform random number between 0.0 and 1.0

round - integer result of rounding the real argument to nearest whole value

trunc - integer result of truncating the real argument to whole part only

float - real result of closest value to the integer argument

The operator-procedure correspondences are as follows: '+' represents iplus and rplus; '-' represents idiff, rdiff, ineg and rneg; '*' represents itimes and rtimes; 'mod' represents imod and rmod; '/' represents idiv and rdiv; '**' represents iipower, rrpower and ripower; '|' represents iabs and rabs; 'sign' represents isign and rsign.

Sample arithmetic expressions:

```
Declarations used:
   real X,Y,Z;
   int I,J,K;
Real expressions:
   X+float 2
   X**I-|(Z+16./(Y-6.))
   (float 12+13/I-12)**2.106E-32
Integer expressions:
   I+J-3
   trunc(float(I)**.3)+iplus(2,6-K)
```

4.7 Strings

ALAI has two kinds of strings, fixed length and variable length. If a string is declared as in 'string(800) LONG', where the bracketed quantity must be an integer constant greater than zero, then the declared string will always be of the given length. If it is assigned a string which is too short, then the short string is padded on the right with blanks to the correct length. If the string to be assigned is too long, then it is truncated on the right. If a string is declared as in 'string VARYING', then it is of varying length. The two types of strings can be freely intermixed. String constants are sequences of characters enclosed in quotes. The quotes can be either both ' or both ". If the quote used as a delimiter is to be a character of the string, then each single occurrence desired is entered as a pair of the quotes.

Substrings can be picked out (as expressions or as references) via substring designators placed after the string expression or reference. These are one or two integer expressions, enclosed in brackets and, if necessary, separated by the character '|'. The first expression is the position within the string of the first character of the substring (the first position in a string is position 1). The second value is the number of characters to take in the substring. If not given, a length of 1 is assumed. Run-time errors occur if this designator attempts to use a character beyond the bounds of the string, i.e. if the first value is less than 1 or if the sum of the two values minus one is greater than the present length of the string. The identifier, *L, used within the substring designator expressions, represents the length of the string.

The following procedures are defined:

sconc - string concatenation

trim - string result is argument with all trailing blanks removed

trim1 - string result is argument with all but one trailing blank removed (one blank is added if necessary)

decode - integer character code of first character in string argument

code - string of length one coded from integer argument

dupl - string result is the first (string) argument duplicated the second (integer) argument times

intget - integer result is the decoded value of the string argument. If the argument is invalid, the value returned is that of the global integer *INTERRV and the global boolean *INTERR is set to true.

realget - real result is the decoded value of the string argument. If the argument is invalid, the value returned is that of the global real *REALERRV and the global boolean *REALERR is set to true.

repfix - string result is the fixed point representation of the first (real) argument of length the second (integer) argument and with the third (integer) argument digits after the decimal point. If the second argument is negative, then the length is the shortest needed to represent the number. If the third argument is -1, then the decimal point is not included. If the third argument is less than or equal to -2, then as many digits as are necessary are included after the decimal point. If the representation cannot be made, then the result is an appropriate length string (3 for second argument negative) of asterisks and the global boolean *FIXERR is set to true. Sample 'repfix' conversions: repfix (2.031,-1,0) => "2.031", repfix (2.031,6,3) => " 2.031", repfix (2.031,6,4) => "2.0310", repfix (2.031,3,-1) => " 2", repfix (2.031,7,-2) => " 2.031", repfix (2.031,6,2) => " 2.03".

repsci - string result is the scientific representation of the first (real) argument with the second (integer) argument digits after the decimal point. If the second argument is negative, the decimal point is not included. The exponent is written as 'E' followed by a sign or blank, followed by two digits (one or both may be zeroes).

repint - string result is the representation of the first (integer) argument with sufficient leading blanks to have a length of the second (integer) argument. If the second argument is negative, then the result is as long as is needed to represent the first argument. If the representation is not possible, then the appropriate number of asterisks is returned and the global boolean *INTRERR is set to true.

The operator '+' can be used to represent sconc. The global boolean *PLUSBLANK controls the insertion of a plus sign for the three 'rep...' procedures. If it is true, a plus sign is not inserted, instead, a blank is used (except when the total length is unspecified, in which case no character is inserted). If *PLUSBLANK is false, then the '+' sign will always appear for positive numbers.

Examples of string manipulations:

```
string (20) A;
string B,C;
string(1) CHAR;
CHAR:="+";
A(13):="only the 'o' of 'only' will be moved in";
B:="now B is of length 21";
C:='string with quotes '' "ok"';
B(16|5):=trim1 A;
C:=B+trim1 CHAR(1|1+3-(9 div |3));
```

4.8 Bits

Bits constants are represented as binary, octal or hexadecimal constants. Binary constants are sequences of '0's and '1's preceded by a '¢'. Both octal and hexadecimal constants use the prefix '|'. When the sequence is decoded, it is interpreted as either octal or hexadecimal, depending on the current value of the fixed boolean *HEX. If *HEX is true, the interpretation is as hexadecimal, if *HEX is false, then the digits must be all octal digits and they are interpreted as an octal constant. The exact number of bits allowed in bits values will depend on the machine of implementation, as will the available accuracy and size of arithmetic values. Bits values can be subsequenced, exactly as with strings. A single bit (picked out from a bits value) is equivalent to a boolean value. When operating with bits subsequences, the lengths must match.

The operators '¬', '&', '|' and '*' represent bitwise negation, AND, inclusive and exclusive OR respectively. The operators 'shl' and 'shr' shift bits values left or right an integer number of positions. The operator '%' will return the bits value of its argument. If the argument is an integer, its binary representation is returned; if the argument is a type which uses a pointer to a language defined record, then the result will be the binary representation of that pointer, etc.

The following procedures are defined:

logand - bitwise 'and' of two bits arguments

logor - bitwise 'inclusive or' of two bits arguments

logxor - bitwise 'exclusive or' or two bits arguments

lognot - bitwise complement of bits argument

shift - bits result is the first (bits) argument shifted left by the second (integer) argument places. If the second argument is negative, then the shifting is to the right.

bits - bits result is the bit representation of the argument, which can be of any type. If it is a pointer, then the bits representation of the pointer is returned.

binget - bits result is decoded value of string of binary digits. Error will set fixed boolean *BINERR to true and return the present value of the fixed bits *BINERRV.

octget - bits result is decoded value of string of octal digits. *OCTERR and *OCTERRV work as *BINERR and *BINERRV.

hexget - bits result is decoded value of string of hexadecimal digits. *HEXERR and *HEXERRV work as *BINERR and *BINERRV.

bitget - bits result is the decoded value of the string as a hex or octal (depending on the value of *HEX) or binary value given complete with initial '¢' or '|'

repbin - string result is full width binary representation of bits argument

repoct - string result is full width octal representation of bits argument

rephex - string result is full width hexadecimal representation of bits argument

bitint - integer result is value of bits argument

The operator '%' is equivalent to the procedure bits. In the three 'rep...' procedures, no prefix ('¢' or '|') is included in the result. The operators '&', '|', '*', and '¬'

are equivalent to logand, logor, logxor and lognot respectively. 'shl' and 'shr' are used to represent shift; if 'shr' is used, then the integer shift amount is negated before passing it to shift. The bits operators precedence scheme is:

```
'%'
'¬'
'shl', 'shr'
'*', '&'
'|'
```

Examples of bits manipulations:

```
bits A,B,C;
A:=¢011010;
B:=!017FC;
C:=A|B&¢1111000000000000;
B(I|J+6):=C(K|J+6);
A(J):=¬C(J-3)|(%13.2)(6);
```

4.9 Booleans

Booleans can take on one of two values, true or false. They are used mostly for flags and condition indicators. The 'if', 'cond' and 'case' constructs use boolean values to control the flow of program execution. Most boolean values are originally produced as the result of comparing two things. The comparison operators, '<', '<=', '>' and '>=' (see later for '=' and '¬='; these have wider application and are therefore not classified as "comparson operators") compare the values of two integers, two reals or two strings. String comparisons compare the integer equivalents (as returned by the procedure decode) of the characters in the strings, the left-most characters being 'most significant'. If the strings are not of the same length, then the short one is padded on the right with characters whose integer equivalent is zero. The integer equivalents of characters will vary from machine to machine, but will most often be either EBCDIC or ASCII, both of which produce acceptable alphabetic orderings when strings are compared.

The identity comparison operators, '==' and '¬==' register equality only if the immediate values of the comparands are the same. For integers, reals, bits and boolean values, these operators test the true values of the comparands, but for all other values, the pointers to the values are tested, and not the values themselves. Thus if one has

```
string A,B,C,D:="days";
A:="happy"+D;
B:="happy"+D;
C:=A;
```

then the following are true:

```
A==C
A¬==B
B¬==C
```

The equality comparison operators, '=' and '¬=' test the actual values of compound values like strings, vectors, stacks, etc. Note that A==B implies A=B and A¬=B implies A¬==B. From the above example one would have

```
A=B
A=C
B=C
```

If the expressions compared are of different types, they will never be either equal ('=') or identical ('==').

As mentioned earlier, a single bit, picked out of a bits value, is a valid boolean expression. Thus (¬¢0)(1) is equivalent to true and ¢0(1) is equivalent to false. Note that no matter how many digits are given in the source for a bits constant, the constant itself always has the full length used in the particular implementation of ALAI. Thus ¢0 and ¢1 themselves are not valid boolean expressions.

The complementation operator ('¬') applied to a boolean yields a boolean which is the complement of the argument. The boolean operators '&', '*' and '|' represent the operations of boolean AND, boolean exclusive and inclusive OR respectively. Note that if the first argument for '&' or '|' is, respectively, 'false' or 'true', then the second argument is not evaluated. The precedence scheme for the boolean operators is as follows:

```
'¬'
'=', '¬=', '==', '¬=='
'*', '&'
'|'
```

Note that the comparison operators cannot have boolean arguments, hence the arguments for such comparisons are always evaluated fully before any boolean operations on the comparison result commence.

The following procedures are defined:

bnot - boolean complement of boolean argument
band - boolean 'and' of two boolean arguments
bor - boolean 'inclusive or' of two boolean arguments
bxor - boolean 'exclusive or' of two boolean arguments

comp - directly compares its first two arguments ('A' and 'B') according to the third (the 'key'). The boolean result is true if
1) key=-2 and A<B or
2) key=-1 and A<=B or
3) key=0 and A=B or
4) key=1 and A>=B or
5) key=2 and A>B
Any other value of the key (an integer) produces false. This procedure does direct signed integer comparison; if passed pointers as arguments, it will merely compare the pointers, hence it can be used to put an ordering on arbitrary values.

realcomp - as comp but requires real arguments

stringcomp - as comp but compares strings, not their pointers

Sample boolean expressions:

```
A<=B&B<=D|A=B&B>D
P|¬(Q*F:=S=="This string")
I¬=3&(B1|¬B2)(6)
```

4.10 Pairs, Triples and Vectors

Vectors are one-dimensional, ordered lists of objects, all of which are of the same type (that type can be 'free', however). A given vector variable may point to vectors of any length or, in the case of fixed sized vectors, the variable actually is the vector. Referencing a vector variable (or an element of a vector vector or vector array, etc.) refers to the entire vector. Individual objects (elements) in the vector can be referenced (either for obtaining their value or for assigning a new value) by indexing the vector with a single integer expression. Indices start at one with the left-most element. Pairs and triples are essentially vectors of two or three elements respectively. Their elements can be selected via indexing as in standard vectors or via their field names in the record classes (language defined) which represent pairs and triples. The elements of pairs are *LEFT and *RIGHT (the left and right elements respectively) and the elements of triples (from left to right) are *LEFT, *MIDDLE and *RIGHT. Thus if A is a triple, 'A@*MIDDLE' and 'A(2)' are equivalent.

Pairs, triples and vectors can be indicated in source language in two ways. The form 'P<A,B>' produces a pair whose elements are A and B (both of which can be arbitrary expressions of the appropriate type). A and B can be omitted (but not the comma) in which case they are assumed to be null (zero for int, real and bits pairs; false for bool pairs). They are parsed in the normal manner. The form 'P(A B)' has similar results except that A and B are treated differently. If A and B are simple expressions like '2', 'A+B-(C div 3)', etc. but with no embedded blanks which are not enclosed in brackets, then the expression result is used in the pair. (In this form embedded blanks are significant - they delimit the elements of the pair.) If A or B is totally enclosed in brackets, then they are interpreted as pairs in the same manner as the original pair. Thus

```
P(3+6 ((1 -2) (6 4)))
```

is equivalent to

```
P<3+6,P<P<1,-2>,P<6,4>>>
```

but

```
P((2 4) 1 - 2)
```

is in error because the outer pair has too many elements (the pair P<2,4>, 1 and -2). Analagous constructs using the pseudofunctions 'T' and 'V' produce triples and vectors respectively. Triples must have three elements, but vectors can have any number of elements. When using the syntax with normal (round) brackets, care must be taken to ensure that any blanks and bracketing used in the elements will not be misinterpreted as delimiters. An expression which must be entirely bracketed (e.g. to enclose some embedded blanks, perhaps around the operator 'mod') must be preceded by a colon to ensure that it is parsed correctly. The colon tells the parser to switch back to normal parsing for the extent of one expression. The special forms null and () are valid pairs, triples and vectors. The form V<> denotes a vector with no elements (an empty vector).

No operators are defined for pairs or triples. Vectors can be concatenated with the '+' operator. Sample vector expressions:

```
V<,,,,>(J-3):=J+3
A+V(Q P () (X+Y+Z))
V(2 .1 :((V(1 2 3 "Cat")+V<Q>)(I+3)) (2 1 0))
```

The following procedures are defined:

vectconc - vector result is concatenation of two vectors

bpair - pair result has the two arguments as elements

btriple - triple result has the three arguments as elements

bvector - vector result has the arguments (if any) as elements pairequal - returns true if the two pair arguments are equivalent

triplequal - as pairequal but compares triples

vectequal - as pairequal but compares vectors

4.11 Sets, Bags, Lists and Chains

Sets, bags, lists and chains are ordered linear sequences like vectors, but the way in which they are ordered and the way in which they are constructed internally makes them significantly different. Internally, vectors are a group of consecutive storage locations pointed to by a field of the language defined record which represents the vector. Sets, bags, lists and chains, however, are represented as linked lists, the pointer to them merely points to the first element of the linked list. It is thus computationally easy to add to, delete from and chain together these objects, whereas to do so with vectors requires the creation of a new vector each time a length-affecting change is made.

Lists are most like vectors in that the order of the elements is whatever order the user used when creating or modifying the list. Thus multiple occurences of elements are not even recognized by the ALAI internal procedures when dealing with lists (and chains and vectors). Bags, however, are sorted by ALAI, the user has control only of the number of duplicates of each item contained in them; they correspond to bags of coloured marbles. It is the quantities that are important, not the ordering. Sets, like bags, are internally sorted by ALAI, but sets can contain only one occurrence of an element. The presence or absence of an element, not its position or number of occurrences, is important. Chains are user ordered like lists, but they are represented internally as doubly linked lists so that they can be traversed in either direction. Lists are built of two-field records whose fields are *VALUE and *NEXT. Chains are built of three-field records whose fields are *PREVIOUS, *VALUE and *NEXT. Using the given field names, lists and chains can be freely traversed and altered by the user through pointers which are of type list or chain respectively.

Sets, bags, lists and chains can all be indexed by a single integer expression. This indexing yields a pointer to the appropriate value in the compound element, not to one of the records from which it is constructed. If A is 'int list' then A(3) is 'int' and not 'int list'. Sets, bags, lists and chains can be specified in source language in ways analagous to those for pairs, triples and vectors. The pseudofunctions used are S, B, L and C respectively. The form '< ... >' can be used without a pseudofunction, in which case a set is produced. The forms null and () are also valid sets, bags, lists and chains.

The set operators '*', '+', and '-', with precedence ordering (highest to lowest)

'*'
'+', '-'

represent the set operations of intersection, union and difference respectively. The set comparison operators '<', '<=', '>' and '>=' represent the respective set comparisons of proper subset (the sets are not equivalent), subset, proper superset and superset. The comparison operators '==' and '¬==', and '=' and '¬=' test set identity (the same set) and set equivalence (the same elements).

The set comparison operators have been extended to apply to bags; bag A is a superbag of bag B if for every element in bag B, there are at least as many occurrences of that element in bag A as in bag B, bag A is a proper superbag of bag B if for every element in bag B, there are more occurrences of that element in bag A than in bag B, etc. The bag operators '|', '&', '+' and '-' produce result bags in which the number of occurrences of the elements is the maximum, minimum, sum or difference respectively of the number of occurrences of the elements in the arguments. With the '-' operator, result counts that would be negative produce a result count of zero and set the fixed boolean *BAGERR to true.

The list and chain operators '+' and '*' cause appending and concatenating of the list and chain arguments respectively. '+' creates a new list or chain consisting of the elements from both arguments and returns it as result; '*' modifies the last element of the first argument (and the first element of the second argument in the case of chains) to point to the other argument and returns a pointer to the modified first argument. The '-' list and chain operator returns the reverse of its argument (the elements appear in the opposite order). The operator '<-' (and its negative '¬<-') tests set and bag membership.

The following procedures are defined:

bset - result is set of the arguments

bbag - result is bag of the arguments

blist - result is list of the arguments in the same order as given

bchain - result is chain of the arguments in the same order as given

intersect - set intersection of two sets

union - set union of two sets

setdiff - set difference of two sets

bagplus - bag result has element counts which are the sum of the corresponding element counts in the two arguments.

bagdiff - bag result has element counts which are the difference of the corresponding element counts in the two arguments. If any element count would be less than zero, then it is set to zero and the fixed boolean *BAGERR is set to true.

bagmax - bag result has element counts which are the maximum of the corresponding element counts in the two arguments

bagmin - bag result has element counts which are the minimum of the corresponding element counts in the two arguments

listconc - attaches the second list argument to the end of the first and returns the modified first argument

listappend - produces and returns a list whose first part is a copy of the first list argument and whose remainder is a copy of the second list argument

chainconc - attaches the second chain argument to the end of the first (complete with the necessary back link) and returns the modified first argument

chainappend - produces and returns a chain whose first part is a copy of the first chain argument and whose remainder is a copy of the second chain argument

listreverse - constructs and returns the reverse of the list argument

chainreverse - returns the reverse of the chain argument

setcomp - returns a boolean value resulting from comparing its first two arguments (sets) according to its third (integer) argument. See comp under 'Strings' for details on the third argument. The sets are compared on the basis of a subset - superset - equivalence relationship.

bagcomp - like setcomp except compares bags (see above discussion for the meaning of bag comparisons)

listequal - boolean result is true if the elements of the two list arguments are the same and in the same order (the lists are equivalent)

chainequal - returns true if the two chain arguments are equivalent

smember - returns true if the first argument is a member of the second (set) argument

num - integer result is the number of occurrences of the first argument in the second (bag, vector, list or chain) argument

Sample set, bag, list and chain manipulations:

```
set S; int bag IB1,IB2; list L; real chain RC;
S:=<3,6,2.45,.CAT,"apple">;
/* 3<-S is true, 4<-S is false */
S:=S+<.DOG,3,"apple">;
/* S will now be equal to:
<3,6,2.45,.CAT,.DOG,"apple">   */
S:=S*S<3,"apple","cat">;
/* yields S = <3,"apple">   */
IB1:=B<2,2,3,4,4,4>;
IB2:=B<1,1,2,4>;
/* IB1+IB2 = B<1,1,2,2,2,3,4,4,4,4>
   IB1-IB2 = B<2,3,4,4>
   IB1|IB2 = B<1,1,2,2,3,4,4,4>
   IB1*IB2 = B<2,4>    */
L:=ANOTHERLIST; /* now traverse it */
while L¬=null do
   L:=L@*NEXT;
/* process the element here */
   od;
RC:=C<3.2,4.7>*OTHERCHAIN;
RC(4):=27.34; /* modify the chain */
for I from 1 to 8 do RC:=RC@*NEXT od;
for I from 1 to 5 do RC:=RC@*PREVIOUS od;
/* now RC@*VALUE = 27.34 */
```

4.12 Stacks and Queues

Stacks (push-down stores) and queues (first in-first out stores) are ordered lists in which access has been limited to certain prescribed methods. Stacks allow access only through one end of the list (the top of the stack); elements are 'pushed' onto the stack and 'popped' from the stack (in the reverse order from which they were pushed on) through this one access point. Queues allow access through both ends of the list; elements are 'added' to the back and removed, or popped, from the front (in the same order as they were added). ALAI implements both stacks and queues as linked lists.

Most stack and queue manipulating conventions require that the top or front element be popped before it can be referenced. This is a nuisance for stacks (it often must be pushed back on) and disasterous for queues (it can't be put back in without cycling through the entire queue). In ALAI, this problem has been alleviated by the following convention: a stack or queue expression prefixed by the operator '<', yields a reference to the top or front element of the stack or queue (respectively). If the stack or queue is empty or uninitialized, the reference is invalid. Thus the top element of a stack and the front element of a queue can be referenced and changed without being removed from the stack or queue. ALAI also allows reference to elements within a stack or queue by indexing. Indexing starts at the top of the stack or the front of the queue with an index of one.

The new statement forms needed to operate stacks and queues are the push, pop and add statements. 'Push' followed by a stack expression will add a new element to the top of the stack with initial value null, zero or false, depending on the type of the stack. It is legal to push onto an empty or uninitialized stack. 'Pop' followed by a stack or queue expression will remove one element from the top of the stack or the front of the queue. Trying to pop an empty or uninitialized stack or queue has no effect but will set the fixed boolean *STACKERR or *QUEUERR to true (respectively). Popping the last element turns the stack or queue into an empty one. 'Add' followed by a block-expression, followed by 'to', followed by a queue expression, will add the value of the first expression to the back of the queue which is the value of the second expression. (The first expression is fully delimited by 'add' and 'to', hence it can be a block-expression.) The operator 'empty' tests the emptiness of stacks and queues. It is used to represent the procedures sempty and qempty. The difference between empty and uninitialized stacks can be seen in the following:

```
stack A,B,X,Y;
A:=B;
B:=emptystack;
push B;
<B:=6;
Y:=emptystack;
X:=Y;
push X;
<Y:=6;
```

X and Y now point to the same stack which has one element (6). B points to a second stack which also has one element (6). A is still uninitialized (null). Similar relationships hold between empty and uninitialized (null) queues. Stacks and queues can be made uninitialized by assigning null or () to them.

The following procedures are defined:

emptystack - returns a unique empty stack

emptyqueue - returns a unique empty queue

sempty - returns true if the stack argument is empty

qempty - returns true if the queue argument is empty

stackequal - returns true if the two stack arguments are equivalent

queuequal - returns true if the two queue arguments are equivalent

push - pushes the stack supplied as argument

addqueue - adds the second argument to the queue supplied as first argument

popstack - pops one element from the stack argument

popqueue - pops one element from the queue argument

4.13 Records

A record is a single structure containing several different but related pieces of information. The pieces of information, called the fields of the record, can be of varying type and number, but are fixed within a given record class. In ALAI, record classes are defined via <record-declaration>'s. Variables of type 'ref' are pointers to records; if the 'ref' is followed by a bracketed identifier, then that identifier must be the name of a record class and the variables being declared can point to (reference) records of that class; if no such identifier is given, then the variables are of type 'ref' (rather than 'ref(<identifier>)') and can point to records of any class. The name of the record class becomes a "procedure" which generates records of that class; if called with no arguments it produces a record with all null or zero fields; if called with arguments, then the argument list must contain one less comma than there are fields in the record class. When called with arguments, the "procedure" returns a record whose fields have been filled in with the values passed as arguments, or if no value was passed for a field, that field is initially null or zero.

Fields of records (normally in ref variables or arrays, etc.) can be accessed by following the ref expresssion by an '@' followed by an integer expression. The integer is used as an index to the record, indexing starting at one with the leftmost (in the <record-declaration>) field. The result can be used in expressions as a value or can be assigned to. The infix '@' operator does not take precedence over indexing or procedure calling. The fields of records pointed to by 'backall' ref variables are made 'back' while the variable points to that record. The value 'null' (equivalently '()') is a valid value for all ref variables and it is distinct from all records.

To simplify the use of records and to increase the amount of parse-time checking that is possible, the declaration of a record class produces integer variables, associated with that class, whose values are initialized to the appropriate index for the fields whose names they bear. These variables can be used by the programmer, but their values cannot be changed. The same identifier can be used in more than one record class, resulting in conflicting values for the resulting integer variable. In such cases, the conflict is resolved by simply disallowing use of the variable except when the parser is sure of the record class in use (either from its own checks or via a user check or guarantee). Thus if the following declarations have been made:

```
   record A(int VALUE; ref(A) PTR);
   record B(int CODE,VALUE,PTR);
   ref X;
   ref (A) Y;
```

then one can have

```
   Y@VALUE:=3;
   (/ref(B) X)@PTR:=4;
```

but not

```
X@VALUE:=bitint(%X@PTR);
Y@CODE:=2+PTR div 2;
```

The left-to-right evaluation rule for operators of equal precedence produces the following result:

```
A@B@C@D = ((A@B)@C)@D  etc.
```

The record class declaration also produces a secure integer (can be referenced but not assigned to) whose name is the same as the record class (and hence the same as that of the record generating procedure for the class) and whose value is equal to the type number which has been associated with the record class. These type numbers are unique to each record class and are an integral part of each record, hence they can be used to identify the record class to which a particular record belongs. The syntax used is that of the boolean expression:

```
<expression) {is | isnt} <int-expression>
```

The left-hand expression can be of any type, but is usually 'ref', 'any' or 'free', i.e. something that encompasses a range of sub-types. The right-hand integer expression is usually a special secure variable such as 'ref' followed by the name of a record class or a user defined data type name or a standard ALAI data type name (a single word only, e.g. 'int' or 'array' or 'stack', not a complex type like 'real list') but can be an arbitrary integer expression. The parser translates occurrences of 'is' (and its negative 'isnt') into calls on the procedure 'type'. e.g.

```
X is Y
```

becomes:

```
type(X)=Y
```

which in turn becomes:

```
comp(type(X),Y,0)
```

Set membership tests can of course be used:

```
type(X)<-<A,B,C,D>
```

No ambiguity in usage of the record class name can occur, since the integer version can be used only as a bracketed term following 'ref'.

The following procedures are defined:

type - integer result is the type of the argument which can be of any type.

recordequal - returns true if the two record arguments are equivalent.

4.14 User Defined Data Types

User defined data types consist of an identifier representing the data type (it is a secure integer whose value is the type number for that particular data type) and a set of identifiers representing the values of that data type. The set of value identifiers is available at run-time so that values of the user defined data types can be read and written as strings. Each user defined data type is represented by a unique type number which is an integral part of any value of such a type. The value 'null' is also a valid value for any user defined data type; its integer equivalent is zero, whereas the user-specified values start at one and go up to however many values are specified. The name of the data type can be used in 'is' or 'isnt' to check the current type of 'any' values. The type 'any' encompasses all user defined data types. No operators are defined for user defined data types. The simple comparisons '=' and '¬=' will compare values from the same user defined data type. The following procedures are defined:

reptype - returns a string which is the identifier associated with the user defined data type value passed as argument.

gettype - returns a valid user defined data type value corresponding to the string passed as the first argument. The second (integer) argument is the type number for the data type or if zero then all data types are searched for the name. If no such value exists, then the fixed boolean *TYPERR is set to true and null is returned.

type - see under 'Records'

Sample user data type manipulations:

```
type DIRECTION (NORTH,SOUTH,EAST,WEST);
type SEXX (MALE,FEMALE,NEUTER,HERMAPHRODITIC);
record PERSON(string NAME; int AGE; SEXX SEX;
   DIRECTION POINT);
int N, TOTAL:=0;
DIRECTION CLASSPOINT; ref(PERSON) SOUL;
while inline(CLASSPOINT); CLASSPOINT¬=null do
/* sample data: " SOUTH " */
   inline(N);
   TOTAL:=TOTAL+N;
   for to N do
      SOUL:=PERSON(,,,CLASSPOINT);
      inline(NAME(SOUL),AGE(SOUL),SEX(SOUL));
/* data: "  'Jones,Harry'  23 MALE " */
   od od;
/* the created records are here just ignored */
```

4.15 Labels

A label is a pointer to a location inside a program to which control can be transferred via a goto statement. Label constants, created by placing an identifier (the constant's name), followed by a colon, before any executable statement are local values. As such they cannot be assigned to any label variable which is more global than themselves. Goto statements, 'goto' or 'go' 'to' followed by either a label constant or a label variable, etc. (i.e. a label expression) cause a branch to the statement labelled by the label constant which is the value of the expression. It is illegal to branch into a block. Branching out of a block causes normal block exiting of that block (any backtrack points previously established in that block will of course still exist). Branching within a block merely changes the order of execution of the statements of the block. No operators are defined for label values.

It has been argued that the 'goto' construct is dangerous in that it causes confusion as to what conditions exist when a given statement is executed. Also, in a sufficiently general programming language, goto's are redundant, i.e. any program can be written without using them. ALAI as a language is meant to be easy to use, however, so the 'goto' construct has been maintained for the benefit of those who are used to using them (e.g. FORTRAN programmers) and for cases in which it is the most natural construct to use. More restricted constructs (less capable of causing confusion) are available in the form of 'exit', 'return' and 'next'.

Sample label manipulations:

```
   label L;
   L:=if A<B then LOOP else END;
   goto L;
   ---
LOOP:X:=3;
   if Y<X then go to END fi;
   ---
END:return
```

4.16 Procvals

A procval variable is a variable which takes procedures as values. Procval constants are produced by prefixing the procedure name with '@'. Procedures which are the value of a procval expression can be called by prefixing the procval expression by '&' and postfixing it with a bracketed argument list (optional if the call uses no arguments). The '&' operator takes precedence over infix operators, argument lists, indexing and field selection.

Sample procval manipulations:

```
   procval P;
   proc THING;
      X:=Y;
   int proc F(int X);
      (3*X+4)*X+5;
   P:=@THING;
   &P;
   P:=@F;
   I:=I+&P(I);
```

4.17 Locations (loc's)

Loc variables have as values pointers to referencable and assignable locations. These locations can be variables, specific fields of specific records, elements of arrays, etc. A loc constant is produced by preceding a valid <location> by '@'. The resulting loc is referenced both for obtaining its current value and for assignment, by preceding the loc expression by '&'. As many levels of such indirection as are needed can be used. Ambiguous cases, such as using '&' before a free value (it could be either a loc or a procval which returned a value), are not allowed. Entire <location>'s are evaluated before applying the '@' operator, thus if X is a ref, then '@X@3' is equivalent to '@(X@3)' but not to '(@X)@3'. (This latter form refers to the third field of the language defined record which represents the loc, i.e. to a non-existant field.) No other operators are specifically relevant to loc's. Loc values cannot be passed out of the range of the variable being referenced, hence the parser will not allow loc expressions to be passed outward other than as arguments in a procedure call.

Sample loc manipulations:

```
record Z(int VALUE; ref(Z)PTR);
loc A;
ref(Z)P;
int list Q;
int array R;
int S;
&(case 1+I mod 4 of (@S, @R(S+4, I div 4), @Q(I-8),
      @P@VALUE):=6;
A:=@S;
```

```
&A:=3+&A;
```

The '@' operator, unlike '&', does not take precedence over indexing, argument lists or field selection.

4.18 Arrays

Arrays are regions of storage arranged as multidimensional rectangular blocks. ALAI places no limits on the size or dimensionality of arrays, but the local operating system or hardware may limit the total size. Arrays have no associated type other than 'array', but array variables can be typed and so can array expressions. For example, if X is declared as 'int list array list array', then the expression 'X(6+I,12)(13-J)' has type 'int list array'. If no basic type other than 'array' is specified, then 'free array' is assumed. Empty arrays returned by the procedure 'array' have no type, and thus can be assigned to any array variable (this is simply a special allowance made by the parser). The argument list for array is also special to the parser in that a non-standard form is accepted. The correct form is any number of pairs of integers, but the parser will accept "pairs" of the form <int-expression> : <int-expression> or simply <int-expression>. In the latter form, the first expression defaults to one. The successive pairs denote the ranges of indices for successive dimensions.

Elements of arrays are valid variables and have the type passed on from the variable which currently points to the array. Elements are selected by following an array expression by a bracketed list of integer expressions, which select along the respective dimensions of the array. The indices must be within the range of the lower and upper bounds passed to 'array' when the array was created. Arrays themselves are global objects, but the values stored in them may not be, hence the globality is controlled through the 'global' and 'local' special modifiers applied to array variables (see 'Declarations'). Individual elements of arrays can be selected as 'loc' values by prefixing an array reference by '@', as in '@A(I,J,K)'. No special operators are defined for arrays.

Fixed size arrays are local, rather than global objects, hence passing of them as arrays necessitates copying the complete array (see 'Declarations'). Fixed dimensionality for arrays merely instructs the parser to check the number of indices used to index the array and the dimensionality of arrays assigned to the fixed dimensionality array variables. This checking is not always possible and serves only to eliminate some run-time checking. The dimensionality of an array expression can be specifically checked or guaranteed by the user through the '/' or '//' typing mechanism using the type 'array' followed by a bracketed integer constant representing the dimensionality.

Sample array manipulations:

```
int array B:=array(3:Z,-16,0);
int C;
real array(*,*,*) D;
bool array(1:3,1:25) F;
bool G;
int loc H;
G:=G&F(1,18+J);
D:=array(I:J,P<13,K+6>,27);
D(I,15,1):=13.2;
H:=@B(20,-6);
&H:=&H+1;
&(H:=@B(I,20-J)):=&H+1;
```

Note the last form, which allows the incrementation of an array element with only one indexing of the array. The left to right order of evaluation requires that H be given its value on the left hand side of the outer assignment.

The following procedure is defined:

array - arguments are a variable number of integer pairs. The result is an array whose dimensionality equals the number of pairs and whose lower and upper bounds correspond to the left and right values of the respective pairs. 'Array' is recognized by the parser and is treated specially (see above).

4.19 Tables

Tables are essentially one-dimensional expandable arrays. They are indexed by 'free' values, i.e. by anything, rather than just by integers. Empty tables (not the same as the null value) are produced by the procedure 'table'. The type of a table refers to the data in the table rather than to the values used to index the table (as with arrays the type is associated with the table variables, rather than with the tables themselves). If a nonexistent index is used, i.e. one for which there is no corresponding entry in the table, then if the reference is used to produce a 'loc' value or as a destination in an assignment, the entry is created and initialized to the appropriate one of null, zero or false; else if the entry is needed to obtain a value, then no entry is created, but the value null is returned and the fixed boolean *TABERR is set to true. Thus new table entries can be created as needed. No operators are specifically relevant to tables.

The following procedures are defined:

table - returns a unique empty table. The table is initially of size equal to the integer argument. If a table must be expanded, then the expansion is in lots of *TABEXPAND entries (*TABEXPAND is a fixed int).

tabarray - converts the table argument into a 2 by (size of table) free array. The indices become the first column and the values become the second column.

arraytab - converts the array argument into a table. The array must be two-dimensional, must have only two columns and no entry in the first column may be duplicated. The first column becomes the indices, the second the entries of the table. An invalid array argument sets the fixed boolean *CONVERR to true and yields a null table. Both arraytab and tabarray do not affect their arguments.

Sample table manipulations:

```
table T1,T2:=table(20);
T2("cat"):="dog";
T2(6):=<6,3,2,1>;
T1:=table(100):
T1(T2("cat")):=P("bow" "wow");
T1(V((6) "cat")):=T2(6);
```

4.20 Areas

An area is an unstructured but contiguous block of storage. The size of the area (the number of words in the block) is an integral part of the area. Area is a simple data type, not a compound type, hence areas cannot be further typed (as can arrays, lists, vectors, etc.). Areas can be indexed by a single integer (the first word of the area is word zero). Words in areas are valid 'free' locations. Words in areas can be used in expressions as 'free' values and can be assigned any values whatever. Assignments within areas are not backtrackable.

The following procedure is defined:

area - returns an area whose size is specified by the integer argument

4.21 Conditional Constructs

The conditional constructs in ALAI are 'if's, 'case's and 'cond's. The statement forms always end in a matching closing word, i.e. 'case' statements end with 'esac', 'cond' statements end with 'dnoc' and 'if' statements end with 'fi'. This convention produces source programs which are slightly easier to read than conventions which do not clearly delimit such constructs. Expression forms of these conditional constructs do not end with a closing word. This allows iteration expressions which are shorter (as most conditional expressions will be). e.g.

```
if A<B then A else B
```

All alternatives in expression conditionals must be of the same type (note that 'free' encompasses all types) and this is the type of the entire expression. Block expressions are allowed in some places, but simple expressions are required in others; a block expression is allowed whenever it is delimited by mandatory keywords or punctuation marks.

The basic 'if' construct, if BOOL then CASE1 else CASE2 fi, is executed in the standard way; CASE1 is executed if BOOL yields true, and CASE2, if present, is executed if BOOL yields false. The 'else' part, here 'else CASE2' can be omitted in the statement form of 'if's, 'case's and 'cond's, in which case the statement is empty if the 'else' part is to be executed. 'If' expressions must have the 'else' part, since an expression must have a value in all cases e.g. if BOOL yields false. The 'elif' parts (elif is a contraction of else if) are optional and of arbitrary number. They serve to provide additional alternatives. In general, the first alternative immediately following a 'then' which immediately follows a boolean which yields true is the one which is executed. If no boolean yields true, then the 'else' part is executed. After a boolean has yielded true, no further booleans are evaluated. The 'elif' can be considered to be 'else if' except that no additional closing 'fi's (iteration statements) are required or permitted.

In the 'case' construct, the alternative executed (evaluated) is selected by one expression, the selector. In the second form of the case, with an integer selector, the value of the selector is used as an integer index to the list of alternatives. The 'else' part, if present, is executed if the selector yields an index which is "out of range", i.e. is less than one or greater than the number of alternatives. In the first form, the selector, here of any type including integer, is compared against the key expressions (the one's before the colons in the case list) and when a match is found, the corresponding alternative (immediately after the same colon) is executed. The key expressions can be arbitrary expressions but will most often be simple constants as in:

```
case PERSON of ("John":26,"Mary":28,"Paul":I-J,
   "Susan":F(K))
```

The comparison used with the key expressions is of the '=' type. Case expressions must always yield a value, but the else part is optional, so it is possible that none of the alternatives are selected. If this happens, null (or zero or false) is returned and the fixed boolean *CASERR is set to true.

The 'cond' construct is equivalent to the 'if' construct except that the 'else' part can be omitted. It does, however, require fewer words. For example:

```
cond A:B,C:D,E:F,G:H else I
```

is equivalent to:

```
if A then B elif C then D elif E then F elif G then H
   else I fi
```

which in turn is equivalent to:

```
if A then B else if C then D else if E then F else if G
    then H else I fi fi fi fi
```

Cond expressions, like case expressions, can fail to evaluate any alternative if the 'else' part is missing. If this happens, null (or zero or false) is returned and the fixed boolean *CONDERR is set to true.

4.22 Iterative Constructs

The basic iterative constructs in ALAI are the iteration statement and expression. The ALAI iteration statement incorporates the features from most FOR, WHILE, DO and SUCHTHAT forms in other programming languages. An examination of the grammar (Appendix A) shows that the simplest form of a iteration statement is 'do <block> od', which does not even contain a 'for'. This form is equivalent to the standard ALGOL block as in say, ALGOLW. It can be used to put several statements where the language allows only one. The form 'while BOOLEAN do <block> od' is a simple iteration in which BOOLEAN, then <block> will be executed repeatedly until BOOLEAN yields false (<block> may not be executed at all). The form 'do <block> until BOOLEAN od' causes <block>, then BOOLEAN to be executed repeatedly until BOOLEAN yields true (<block> will always be executed at least once). If both 'while' and 'until' are used, then the three are repeatedly executed (in the same sequence as they appear) until either cutoff becomes effective.

'For' parts cause iteration through a sequence of values. The form 'from' - 'to' - 'by' steps through an arithmetic sequence of reals or integers. The three parts must all be either real or integer, the three may appear in any order and any combination of the three may be omitted, in which case defaults are used. The defaults are: from 1 (1.0) by 1 (1.0) to infinity. If the 'to' part is omitted, then that 'for' does not limit the iterations, it merely sets up a counter. If the various expressions (which are evaluated only once, immediately before the iterations commence) are such that the count variable (the <location>, which also is evaluated only once) never actually equals the limit, then the iterations stop with the last value which is within the limits. The 'suchthat' term provides a way of continuing the iteration, but skipping the execution of the main <block>; if any 'suchthat' expression yields false, then the execution of the <block> (or the 'collect' part for iteration expressions) is skipped for that iteration, but all iteration parts are executed (unless a cutoff occurs) and have their normal effect. 'While' and 'until' can also occur with 'for's in which case they interact with the 'for's in such a way that any cutoff is immediately effective.

The 'in' form, without the '@' iterates through the elements of a compound value. For example, if the 'in' expression yields a set, then the <location> successively takes on the members of the set as value. Such iteration through a bag repeats the value the appropriate number of times. Arrays are iterated by changing the last coordinate most rapidly. If '@in' is used, then the <location> is assigned successive 'loc' values which point to the elements of the compound value. For example, if X is 'real loc' and Y is 'real array', then

```
    for X @in Y do &X:=0.0 od
```

sets all of Y to zeros. One advantage of this form is that there is no need for subscript checking since the subscripts are supplied internally. The 'via' form uses as iteration values the results of the 'via' expression, which is usually a generating procedure of some kind. A cutoff is produced by the 'via' if the expression returns null, zero or false.

If more than one 'for' part is used, then they are executed sequentially in the order in which they appear. A cutoff will prevent the execution of any 'for' parts, 'while' or 'until' following the part which produced the cutoff. The various 'for' parts thus operate all at once, but have an ordering as to their execution.

A sample iteration statement:

```
   set B,P,Q,Y,Z; list R; int PP,COUNT;
   for X in Y suchthat X¬<-Z
   for A in B suchthat A¬<-Z
   for COUNT
   while PP rem COUNT¬=1
   do P:=P+<X>;
      Q:=Q-<A>;
      R:=R*L<P<X,A>>
   until X<-B | A<-Z
   od
```

This particular iteration statement adds certain elements to the sets P and Q and to the list R, but otherwise has no easily discernible practical use. It does serve, however, to illustrate the form of the various components and how they fit together. The statement is equivalent to the following:

```
   set B,P,Q,Y,Z; list R; int PP,I,COUNT,XC,BC;
   bool F;
   YC:=size Y; BC:=size B;
   COUNT:=0; I:=1;
   goto LOOP1;
LOOP:I:=I+1;
LOOP1:F:=false;
   if I>XC | I>BC then goto OUT fi;
   X:=Y(I);
   if ¬(X¬<-Z) then F:=true fi;
   A:=B(I);
   if ¬(A¬<-Z) then F:=true fi;
   COUNT:=COUNT+1;
   if ¬(PP rem COUNT¬=1) then goto OUT fi;
   if F then goto LOOP;
   P:=P+<X>;
   Q:=Q-<A>;
   R:=R*L<P<X,A>>;
   if ¬(X<-B|A<-Z) then goto LOOP fi;
OUT:
```

Iteration expressions are mainly useful for constructing linear compound objects such as sets and lists. The 'do <block> od' of the iteration statement is replaced by a 'using' and a 'collect' part. The 'using' part is evaluated once to produce a procedure which must accept two arguments of the same type and return a value. The type of the arguments must be the same as the type of the 'collect' expression since this expression provides the arguments. The type of the procedure's result determines the overall type of the iteration expression. When a iteration expression is encountered in an executing program, the various expressions which are to be evaluated only once are evaluated (in the order in which they appear) and their results are used throughout the life of the iteration expression. The resultant 'using' procedure is then called with both null (or zero or false) arguments. The value so obtained is the initial value for an internal variable which is used to "collect" or accumulate the final result. On successive iterations through the iteration expression, the 'collect' expression is evaluated and then passed as second argument to the 'using' procedure. The internal accumulator is passed as the first argument and the result is assigned back into the internal accumulator. This process is done once for each iteration through the iteration expression and the last value in the internal accumulator is the final value of the entire iteration expression. For example, the following iteration expression yields a set consisting of the first non-null values produce by the procedure GEN:

```
for X via GEN using union collect <X>
```

Within the iterating parts of the iteration expression (those parts which are executed every iteration, e.g. 'while', 'until' parts) the internal accumulating variable can be referenced through the name '*A'. Two departures from this standard evaluation procedure are made. If the 'using' procedure is the constant 'band' or 'bor' (boolean AND or boolean OR), then a cutoff is generated if the 'collect' expression ever yields 'false' or 'true' respectively. This value is then the result of the iteration expression. The definitions of the following procedures are extended to make them valid 'using' procedures: iplus, itimes, rplus, rtimes, sconc, logand, logor, band, bor, vectconc, intersect, union, bagplus, bagmax, bagmin, listconc, listappend, chainconc, chainappend.

The <location> in the 'for' part is treated specially in that it need not be declared. If it is declared, then the existing identifier is used (it will thus maintain the last value used in the iteration expression), but if it is not, then it is "declared" local to the entire iteration statement or expression. If the parser can determine the correct type to use, then that type is used, otherwise type 'free' is used. The suitable type can be determined from the type of the 'from', 'to' and 'by' expressions (if all three are omitted, then the type is integer) or from the sub-type of 'in' or 'via' expressions.

Sample iteration expressions:

```
/* sum of positive elements of int vector IV: */
   for I in IV suchthat I>0 using iplus collect I
/* inner product of vectors A and B: */
   for R1 in A
   for R2 in B
   using rplus
   collect R1*R2
/* list of first 20 prime numbers */
   for I from 2 suchthat
      (for J in *A
      using band
      collect I div J ¬= 0)
   for K to 20
   using listconc
   collect L<I>
```

4.23 Return, Exit, Iterate and Next

The normal method of returning from a procedure is to execute the last statement in the main block of the procedure body. Sometimes, however, it is desirable to return from some point in the interior of the procedure. This can be done by branching to a labelled empty statement at the very end of the body, but this method can be slightly confusing, especially if the label name is chosen poorly. A more explicit alternative is the 'return' statement which, on execution, causes an immediate return from the procedure in which the 'return' appears. If the procedure being returned from is a typed procedure (i.e. returns a value), then the 'return' must be followed by an expression of the appropriate type which becomes the final value of the procedure call. Such returns can cause several block exits, but do not affect backtrack points.

Specialized goto's are also useful in conjunction with iterations. 'Exit' followed by an identifier which is used as a block name for the body of a iteration statement, will cause that block to be exited immediately, and the iteration statement to be given an immediate cutoff. If the identifier is omitted, then the iteration statement exited is the most local one which encompasses the exit statement. 'Iterate' works similarly to 'exit' except that instead of exiting the entire iteration statement, only the body block is exited. The iterations and tests are then performed and execution continues with the start of the body block. ('Iterate' is essentially a 'goto' to the end of the named body block.) 'Next' is similar to iterate except that after performing the iterations and cutoff tests, execution continues after the 'next' rather than at the start of the body block.

Sample 'exit', 'iterate' and 'next':

```
for I to 5 do
   outbuff(I);
   exit;
   outbuff(-I) od;   /* prints:
1                           */
for I to 5 do
   outbuff(I);
   iterate;
   outbuff(-I) od;   /* prints:
1 2 3 4 5                   */
for I to 5 do
   outbuff(I);
   next;
   outbuff(-I) od;   /* prints:
1 -2 3 -4 5                 */
```

4.24 Input and Output

Input and output (I/O) in ALAI is done via a set of special statements and procedures. 'Card' is the basic input statement; it will take any number of arguments and will read values for each of them from the standard input stream. Statements occurring within the argument list will be executed at the appropriate time. The reading starts with a new input record (card) and reads as many records as are needed (several values may be read from one record, but no value may cross a record boundary) until all of its arguments, each of which must be a valid <location>, has been given a value. A fixed integer, *CARDLEN, tells the input routines how many columns of each record to consider. The external representations accepted are those accepted in program source. The data elements being read must be separated by blanks or commas. The types which can be read in this way are: ints, reals, bits (as bin, oct or hex), strings, user defined data type values and linear sequences of such (i.e. sets, lists, chains, bags, vectors, pairs and triples) represented in either '<>' or '()' format. Linear sequences of linear sequences of linear sequences etc. are allowed and the elements of a sequence can span multiple records so long as no basic element attempts to do so. Free values cannot be read, hence the level of nesting for input of sequences is limited by the level of declaration of the location being read into.

'Inbuff' is a statement form similar to 'card' but which will accept only one argument and which will not cause a new record to be started unless the previous one is exhausted. The procedure 'tercard' (terminate card) simply causes a new input record to be read. The procedure 'getcard' returns a string result which is the next complete record in the input stream (its length need not equal *CARDLEN). Use of 'getcard' does not affect the buffering done by 'inbuff' and 'card'. Output is done in a similar manner except that greater control over format is available. The statements 'line' and 'outbuff' correspond to 'card' and 'inbuff' and the procedures 'terline' and 'putline' correspond to 'tercard' and 'getcard'. When integers and real numbers are being output, the formatting is done by the procedures 'repint', 'repfix' and 'repsci' which were discussed previously (see under 'Strings'). The second and third arguments for 'repfix' and the second arguments for 'repsci' and 'repint' are, respectively, the fixed integers *F_L (fixed length), *F_D (fixed decimal), *S_D (scientific decimal) and *I_L (integer length). The fixed boolean *SCI controls which of scientific or fixed point notation is to be used for real numbers (true -> scientific). For bits values, *BIN decides between binary and hexadecimal or octal (true -> binary) and *HEX, the same variable as is referenced for reading bits values and for use by the bits decoding/encoding routines, decides between hexadecimal and octal (true -> hex). The formats used are those of full length returned by 'repbin', 'repoct' and 'rephex' (see under 'Bits'). The fixed ints 'IPOS' and 'OPOS' indicate the next positions in the buffers to be used by 'inbuff' and 'outbuff' respectively. The procedures 'tabi' and 'tabo' will cause tabbing (skipping of characters or insertion of blanks) into the respective buffers to the position passed as argument.

'Putline' outputs its single string argument without affecting the buffers used by 'outbuff' and 'line'. Free values can be output so long as the actual value being output is of one of the basic types listed above. When printing is being done, some operating systems support carriage control of some kind to allow such things as overprinting, double spacing, page alignment, etc. The fixed variable *CC is meant for this purpose, but because of varying systems, its nature and effects cannot be part of the language specification without unnecessary complications. Similarly, I/O to other than the standard input and output devices will vary in nature, but the procedure names 'read' and 'write' are intended for this use. It is expected that some simple means for the input (parsing) and output (pretty-printing) of expressions under user program control will be provided, but the exact nature of these will not be known until an initial ALAI implementation is produced and in fact may vary from place to place. The following procedures are defined and have been described above:

tercard, terline, getcard, putline, read, write

Sample input/output:

```
int I; real R; string S; list L; bits B;
card(I,R,S,L); /* possible data is:
26    -15.7E21  "Hello " L(2 ("car" 4) 3)    */
/* the same effect is achieved by: */
tercard; inbuff(I); inbuff(R);
inbuff(S); inbuff(L);
/* or by:            */
card(I,R,S); inbuff(L);
/* the above data line could be printed by: */
*I_L:=2; *SCI:=true; *S_D:=1;
line(I,"    ",R,"  ",S,"  ",L);
*BIN:=false; *HEX:=true; B:=!FACB0781;
line(¢101," ",B); /* produces:
00000005 FACB0781           but */
*HEX:=false; line(¢101," ",B); /*produces:
00000000005 37262603601       assuming 32 bit words*/
```

4.25 The Data Net

The ALAI data net is constructed of dictionary entries and user defined record classes. Instances and concepts are created by adding certain fields to the ones in the record classes. These fields are fixed and in fixed positions in the records so that the ALAI net manipulating routines have some predetermined fields to work with. Any record class can be used as a data net node class; thus the user has a high degree of control over net contents. The predefined type 'dict', used for dictionary entries, consists of a string, *NAME, and a list, *MEANINGS. *MEANINGS is a list of, among other things, pointers to net elements which "represent" the entity named by the string *NAME. Two other predefined record classes are used by the system. 'Simple' is a record class which has no fields. 'Expr' (previously considered as a data type due to special considerations by the parser) is a record class with the following fields: dict *REL; free *PAR1, *PAR2, ... , *PARn. Records of this class (and only of this class) can have an arbitrary number of fields. This is allowed because all things which are used as *REL's have an integer on their *MEANINGS list which indicates the number of participants used with them. Thus internal program structures consist of nodes (records) of class 'expr'.

The postfix 'I' (for instance) applied to a record class name yields the name of a record class having the following extra fields: int *PARCNT; I list *PARTIC. (The type 'I' refers to any such record, i.e. any instance.) The postfix 'C' (for concepts) applied to a record class name yields the name of a record class having the following extra fields: int *PARCNT, *USECNT; I list *PARTIC, *USEAGE. The type 'C' refers to any such record, i.e. any concept. The postfix 'N' (for node) yields a class containing both concepts and instances of that class and the type 'N' refers to any concepts or instances. Thus if 'NODE' is a valid record class, then 'ref(NODE)', 'ref(NODEI)', 'ref(NODEC)' and 'ref(NODEN)' are all valid types for net elements.

The extra fields added to instances and concepts are used by the net maintenance routines but can also be used by the user. *PARCNT is a count of the number of times this element (instance or concept) has been used (pointed to) as a field of an instance which was constructed by the net maintenance routines. *PARTIC is a list of the instances in which this element has been used. This count and list does not include the usages of a concept as the concept (the "verb" of a "sentence") in a compound element. *USECNT and *USAGE are an integer count and instance list which include only these latter uses. Thus only concepts can occur as the first component in a compound element if the standard linkup routines are to be used properly.

As an illustration, consider one possible scheme for representing predicate calculus expressions in net form. Simple items (those whose existence is assumed) are represented by the record classes 'ITEMI' (instances) and 'ITEMC' (concepts). Simple propositions are represented by the record class 'COMPI' (they are compound instances). Note that, as with simple record classes, the name of the record class is a procedure which builds records of that class with its arguments as fields. *PARTIC and *USAGE back links are automatically produced for all instances and concepts, and dict records are automatically produced and inserted if strings are passed instead of 'dict' values. Thus we could have:

```
record ITEM(dict NAME);
record COMP1(C PRED; N SUBJECT);
ref(ITEMI) INST1,INST2;
ref(COMP1I) PROP1,PROP2;
ref(ITEMC) CONC1,CONC2;
```

To create items 'John' and 'Sam' and predicate 'Human':

```
INST1:=ITEMI("John");
INST2:=ITEMI("Sam");
CONC1:=ITEMC("Human");
```

Note that the added fields for concept and instance are not given; they are automatically set to zero and null. Also, this use of 'dict' values (in concepts and instances) automatically puts those concepts or instances in the *MEANINGS list for the 'dict's. To state that John and Sam are human:

```
PROP1:=COMP1I(CONC1,INST1);
PROP2:=COMP1I(CONC1,INST2);
```

The *PARTIC lists of the nodes for John and Sam will now have one entry each and the *USAGE list of the node for Human will have two entries. These operations can be combined to allow the easy "stating" of propositions:

```
COMP1I(ITEMC("Pretty"),ITEMI("Sally"))
```

More sophisticated propositions can be handled:

```
record C2(C PRED; N SUBJECT,OBJECT);
record CN(C PRED; N list ARGS);

C2I(ITEMC("Thinks"),INST1:=ITEMI("Bill"),CNI(ITEMC("And"),
   L<COMP1I(ITEMC("Handsome"),INST1),
     COMP1I(ITEMC("Smart"),INST1)>))
```

In this last assertion, one of the fields of a node being constructed was not another node, but rather was a list of nodes. In this case and in others like it, the nodes which are subparts of a field (not itself a node) of a node being constructed are all backlinked to the new node.

Descriptors are concepts or instances for which the net maintenance routines will not backlink from non-descriptor components. These types are represented by a further postfix of 'D' used after (or instead of) the 'C', 'I' or 'N' postfixes. Thus, for the record class 'NODE', the following extra node types are defined: 'ref(NODED)', 'ref(NODEID)', 'ref(NODECD)' and 'ref(NODEND)'. Using the above record classes, one could have:

```
ref(CONC1ID) T1,T2;
T1:=CONC1ID(ITEMC("Happy"),INST1);
T2:=CONC1ID(ITEMC("Superior"),INST1);
```

T1 and T2 will not be backlinked to from 'Happy', 'Superior' or 'Bill' and thus can be efficiently destroyed (deleted from the net) if they prove to be untrue or unnecessary. Note however, that if the following is "asserted":

```
C2ID(ITEMC("Knows"),INST1,T1)
```

then T1 will be backlinked to the newly created descriptor node since it is itself a descriptor. Note that there is no difference between net uses of 'ref(NODE)' and non-net uses of 'ref(NODE)', thus all uses of an instance or concept in any non-descriptor record will be backlinked. The predefined record class 'simple' has no fields, thus the type 'ref(simple)' is a distinguishable pointer to nothing. Such usage is entirely permissible; it is then the value of the pointer itself which is of interest, and not anything the pointer may happen to point to (if anything).

The predefined record class 'expr' is of special interest because of its importance to the ALAI runtime system. It is used for all noncompiled program structures, including patterns. Because of this, it is desirable to have a special, simplified syntax for it. The record class name 'D' is considered to be a synonym for 'exprD' and similar in syntax to the typing pseudofunctions 'S', 'B', etc. This means that it can be used with angle brackets and commas, as in

```
D<"*ASSIGN",X,Y>
```

or with round brackets and no commas, as in

```
D(*IF (*REF :tlookup("X") null) :TRUEPART null :ELSEPART)
```

This latter form is then equivalent to:

```
exprD("*IF", exprD("*REF", tlookup("X"), null), TRUEPART,
   null, ELSEPART)
```

In the latter form, simple identifiers are treated as dictionary entry references to entries with that identifier as name, and normal parsing is achieved through the break character, ':'. Further special syntax for the 'D' pseudofunction is discussed in section 4.27 (Patterns and Pattern Matching).

Any record containing a 'dict' field can be output; the string associated with the 'dict' will be used. 'Dict' values can be created by prefixing a character sequence by a period ('.'). Two procedures are defined for use with data net nodes; they are given below. Also usable is the procedure 'recordequal' (defined in section 4.13) which will test for the equivalence of nodes which are, in actuality, special records. If it is desirable to reference a given node repeatedly (as was largely avoided in the above examples), the most efficient method is to create the node at parse time and use a parse time variable (an 'equ') to reference it as a constant. Run time references to already existing nodes can be obtained via the procedures 'lookup' and 'get'. Note that the operators 'is' and 'isnt' are useful for work with the data net.

The following procedures are defined:

lookup - the string passed as argument is looked up in the current net dictionary. If it is found, the dictionary entry ('dict' record) is returned, else null is returned.

get - the *MEANINGS list of the first (dict) argument is scanned for meanings of the type specified by the second (integer) argument. A list of the meanings found is constructed and returned. The values for the second argument will most likely derive from the predeclared safe integers such as 'int' and 'list' or from record class references such as 'ref(NODE)', etc.

4.26 Unevaluated Expressions and Parse Time and Compile Time Evaluation

A program or procedure in ALAI is processed three distinct times: it is parsed, it may be compiled, and it is executed. Normally, the expressions and statements of a procedure are evaluated or executed when the procedure as a whole is executed. If, however, an expression is prefixed by the character '#', then it is not evaluated to yield a result, rather it IS the result, a result of type 'expr' (expression). Such unevaluated expressions can be passed around, saved, etc. in variables of type 'expr' or type 'free'. Evaluation of such expressions can be forced by preceding an 'expr' expression (an expression whose result is of type 'expr') with the character '='. The result of evaluating the 'expr' then becomes the value of the '=' operator. 'Expr' is a compound type, hence 'expr' variables can be typed further as in 'int expr'. Note that because of their internal structure 'expr's are all 'D's. An 'expr' can be a block expression, but it is illegal to branch out of that block or to fail out of it. References in the 'expr' to local variables are via 'loc' constants created for them; thus 'expr's can be passed to procedures and the variables they reference may thus be used outside of their range. The entire 'expr', however, cannot be passed out of the range of the most local variable it references.

If a statement in ALAI source is preceded by the character '+', then that statement is executed at parse time and does not appear in the resulting internal program structure. Such statements can reference and change 'equ's available in the current block and any fixed variables or data net elements. They are mainly intended as a way of allowing a type of macro (the statement can be a call to a procedure which modifies the program structure produced by the parser) and as a way of allowing the changing, in the middle of parsing, of any fixed variables or data net elements which the parser is dependent on. In a similar manner, and for similar reasons, a statement preceded by the character '¢' is executed at compile time. If the program structure containing a compile time statement is ever interpreted, then the compile time statement is skipped over. Compile time statements can only reference fixed variables and net elements.

To facilitate the locating of parts of programs, the notion of a locator (not to be confused with 'loc's) is introduced. Locators are descriptive ways of writing calls to the procedure 'locate' which returns a type 'expr' pointer to a part of a procedure. Locators are written as a sequence of procedure or block names followed by a sequence of statement numbers, all separated by '#'s. For example 'alpha#beta#12#3#16' refers to the 16th substatement of the 3rd substatement of the 12th statement in procedure (or block) 'beta' which is found inside procedure (or block) 'alpha'. Since most uses for locators in user procedures are those where programs are to be dynamically modified, locators are parsed as calls to 'locate' in all cases, rather than being evaluated at parse time if possible. The exact correspondence between locators and internal program structure will become clearer after the discussion of internal program structures and after direct use of an ALAI implementation which uses locators as parsing prompters (see later discussion on possible operating environments).

The following procedure is defined:

locate - first argument is a list of strings, second is a list of integers. The lists are used as the procedure or block names and the statement and substatement numbers to locate parts of existing programs. An 'expr' pointer to the located structure is returned. If the appropriate structure cannot be found, then null is returned.

4.27 Patterns and Pattern Matching

Pattern matching in ALAI matches specific patterns against other structures or entities. Patterns are constructed of descriptors, hence 'pat' (for pattern) is a subtype of 'D'. Hence also, the syntax for patterns is part of and must be compatible with that for descriptors. Patterns are built up from subpatterns using the pattern operators '¬', '|', '&', '*', '@', '¢', '<-', '<->', '->', '-', '--', '---', ':<-', ':<->', ':->', '=' and '@='. The operators' formats and meanings are as follows (square brackets delimit optional parts):

'¬' - this prefix indicates that a subpattern must not be matched

'|' - this infix indicates that either or both subpatterns must be matched

'&' - this infix indicates that both subpatterns must be matched

'*' - this infix indicates that either but not both subpatterns must be matched

'@[<int-expression>][C=<location>]' - this postfix is used only in patterns representing linear sequences of elements, e.g. instances, strings, lists, sets, etc. If the integer expression is absent, then the postfixed subpattern can appear any number of times (the entire sequence is taken whereas 'arbno' takes only as many as necessary) and the count of occurrences is set into the location (if given). If an integer expression is given, then the subpattern must occur that number of times (negative implies zero). The action is similar to the SNOBOL 'SPAN' pattern. Note that the integer expression and the location are evaluated every time the postfixed pattern is matched (they are evaluated before the matching).

'¢[<int-expression>][C=<location>]' - this postfix is like '@' except that without an integer count it takes as few elements as possible, i.e. it is like SNOBOL'S 'ARBNO'.

'<-' - this prefix indicates that the prefixed pattern is to be evaluated and matched only when the matching process backs up over it

'<->' - the prefixed pattern is to be evaluated and matched when passed over in either direction

'->' - the prefixed pattern is to be evaluated and matched only when passed over in the forward direction

'-' - equivalent to 'arb'

'--' - equivalent to 'span(arb)'

'---' - equivalent to 'arbno(arb)'

':<-' - the true or false result of the prefixed boolean expression is equivalent to the success or failure of matching a pattern prefixed by '<-'

':<->' - as ':<-' but corresponds to '<->'

':->' - as ':<-' but corresponds to '->'

'=<location>' - the portion of the target which is matched by the postfixed pattern is assigned to the location. The assignment is done every time the pattern is matched.

'=@<location>' - assigns to the location the position in the current linear sequence of the postfixed pattern

In a pattern, simple constants stand for themselves and non-pattern expressions (perhaps preceded by a disambiguating colon) are evaluated once, before the matching starts and thereafter their results are constant. Patterns which are unprefixed are assumed to have the prefix '<->'. Postfixes which follow no pattern are assumed to follow the pattern '-'. The location following the '=' postfix (which itself must follow any '@' postfix) is treated as a segment variable, e.g. if a list is being matched, then multiple elements matched by the '@' operator are assembled into a list before being passed on to the location. Note that ':->', ':<->' and ':->' prefix boolean expressions, not patterns. The safe fixed variable *CURSOR points to the unit in the possible target structure which is currently being matched.

The pseudofunctions S, B, V, L, C and D can be used in patterns but the internal structure produced is different from that produced in normal usage. Since patterns are descriptors, they always start with the pseudofunction 'D'. Sample patterns and their meaning:

```
D(string(('B' | 'R') ('E' | 'EA') ('D' | 'DS')))
```

matches 'BED', 'BEDS', 'BEAD', 'BEADS', 'RED', 'REDS', 'READ' or 'READS'.

```
D(V(X | (BIG BROTHER) (-@3)=Y 6))
```

matches a vector containing:

1) the concept X or the instance (BIG BROTHER) followed by
2) 3 of anything; the three are assembled into a vector and assigned to the program variable Y
3) the integer constant 6

```
D(L(string(("h" --- "s") & (--- "ea" --- ))=WORD
    get(lookup("DOG"))(1)@=DOGS))
```

matches a list containing:

1) a string which starts with "h", ends with "s" and contains "ea" somewhere in its interior. The string is assigned to the variable WORD.
2) an arbitrary number of occurrences of the node DOG. The occurrences are assembled into a list and assigned to the variable DOGS.

```
X:=D(string("a" (null | :=#X) "b"))
```

This is a recursive pattern which matches a string consisting of one or more "a"'s followed by the same number of "b"'s. A better way (in terms of efficiency) of accomplishing the same thing is the pattern

```
D(string("a"@C=I "b"@I))
```

where I is an integer variable.

'String' is one of several special "functions" which modifies the pattern matching or provides for special purpose patterns. The names of these pseudofunctions correspond to the names of the ALAI data types they are used to produce patterns to match. The name alone represents any object of that type. 'S', 'B', etc. can be considered as abbreviations for 'set', 'bag', etc. Inside the "argument list" for these functions, the pattern 'arb' represents an element capable of being a part of the type corresponding to the pseudofunction. Inside 'string', 'arb' represents any single character; inside 'bits' it represents any bit; inside 'bool' it represents 'true' or 'false'; inside 'int' or 'real' it represents any numeric value. In more general compound types, e.g. 'list', it represents anything. Inside 'int' and 'real', subpatterns can be comparisons which must be true for the value to be matched, e.g.

```
int(<2 & >I | >2 & >=J)
```

If no such name is used for the global type, then 'list' ('L') is assumed.

The actual forms used for pattern matching are the procedure 'match' and the 'find', 'find1' and 'reset' constructs. The 'find' form searches for occurrences of the <pat-expression> such that the 'with' patterns exist and the 'suchthat' expression is true. The searching is done in the world specified (defaults to the current world). The *PARCNT and *USECNT fields of nodes are used to reorder the search in order to decrease expected search time whenever this can be done without changing the effects of the search. The result of the find expression can be either a pointer to the found target structure (which matches the object of the search, i.e. matches the expression following the word 'find' or 'find1') or a part of the found structure. If the search object contains a value assignment (via the '=' pattern postfix) to the special program variable '*R', then the value of the 'find' is the latest value of '*R'. If 'find' is called with the same object and 'with's as a previous call in the same process, then another example of the object is found and returned. When no more candidates are found (this can happen on the first call) null is returned. Note that 'find' is thus a valid 'via' expression for iterative constructs. The finding of successive values can be reset (and the associated storage space freed) by issuing a 'reset' with the same object and 'with's.

The construct 'find1' is the same as a 'find' immediately followed by a corresponding 'reset' - it finds one example only. If 'need' is specified, then the finds will find the closest candidate not already chosen if no exact match exists. The fixed variable *NEED is then assigned a list of pairs, the left element of which is a small part of the object pattern not matched and the right element of which is the corresponding part of the returned candidate which was "close to" the required form. A sample find:

```
find1 D(DOG I=*R) with D(BLACK *R) with D(BIG *R)
```

Note that it is possible to produce recursive patterns which do not use unevaluated expressions by circularly modifying the internal structure of the pattern.

The following procedures and safe variables are defined:

match - returns true or false depending on the success or failure of the matching of its two arguments. If both arguments are patterns, then the second is treated as a constant and the first is used as a pattern to match against it. If one argument is a pattern then it is matched against the other. Any other combination of arguments is invalid and returns false. The symbol '<>' can be used to represent 'match'.

cright - moves *CURSOR right one element in the current sequence

cleft - moves *CURSOR left one element in the current sequence

cdown - moves *CURSOR down from the current element to its first subelement

cup - moves *CURSOR up to point to the entire sequence of which its previous value was a part. These four cursor moving procedures allow the user to write pattern matching procedures which can be called from within patterns.

arb - a pattern which matches anything. The range of arb can be restricted by pseudofunctions which specify pattern types (e.g. 'string').

arbno - a pattern which will match as many occurrences of its argument as are needed to complete the entire pattern match

span - a pattern which matches any number of occurrences of its pattern argument. It will match as many as there are, regardless of what further parts of the pattern may need to match.

break - a pattern which matches everything in a linear sequence up to something which matches its argument

to - 'to(PAT)' will match as 'break(PAT) PAT'. A value assignment from 'to' yields only the portion of the target matched by PAT.

abort - a pattern which causes immediate failure of the entire pattern match

fail - a pattern which causes failure when encountered

succeed - a pattern which is always matched (to nothing)

tab - matches sufficient elements to cause the next pattern in sequence to be matched against the n+1st element where n is the integer argument to tab

rtab - matches sufficient elements to leave only n elements in the current linear sequence, where n is the integer argument to rtab

len - matches the next n elements of the current linear sequence, where n is the integer argument to len. Tab, rtab and len fail if they cannot perform their normal function.

pos - fails if the next element to be matched in the current linear sequence is not the n+1st element of the sequence, where n is the integer argument to pos

rpos - fails if the next element to be matched in the current linear sequence is not the nth element from the end of the sequence, where n is the integer argument to rpos

Sample program for data net and patterns:

```
 int I; bool QS,QO;
 string S,SSUB,SVERB,SOBJ;
 record R(dict NAME);
 ref(RC) VERB; ref(RI) SUB, OBJ;
 record S(ref(RC) V; ref(RI) S,O);
 ref(SI) SENTENCE; D FORM;
 pat ART:=D(string(("the"|"a"|"some"|"an")@ " "@));
 pat W:=D(string(break(" "))), B:=D(string(" "@));
 while S:=getcard ¬= "stop"
 do if D(string(ART W=SSUB B W=SVERB B ART W=SOBJ))<>S
    then SUB:=RI(SSUB);
       VERB:=RC(SVERB);
       OBJ:=RI(SOBJ);
       QS:=SSUB="who"|SSUB="what";
       QO:=SOBJ="who"|SOBJ="what";
       if ¬QS&¬QO
       then ignore SI(VERB,SUB,OBJ)  /*assert the fact*/
       else FORM:=if QS&¬QO
             then D(VERB - OBJ)  /*looking for subjects*/
             elif ¬QS&QO
             then D(VERB SUB -)  /*looking for objects*/
             else D(VERB - -);  /*looking for both*/
          for SENTENCE via find FORM
          for I
          do line(">",SENTENCE@S,SENTENCE@V,
             SENTENCE@O,".")
          od;
          if I=0 then line(">none found.") fi
      fi
   else line(">invalid input.")
   fi
od;
/* a sample session: */
cat is animal
a dog is an animal
what is an animal
>cat is animal.
>dog is animal.
who is person
>none found.
John is a person
John is what
>John is person.
does Mary have sheep
>invalid input.
bicycle is vehicle
what is what
>cat is animal.
>dog is animal.
>John is person.
>bicycle is vehicle.
a car has some wheels
```

```
a bicycle has wheels
what has what
>car has wheels.
>bicycle has wheels.
stop
```

4.28 Contexts

Contexts (type 'context') in ALAI are used for "snapshots of the world" for use in backtracking and the processing of alternatives. Other languages use constructs called "contexts" as storage bins for variable bindings, i.e. as places in which the programmer can specifically save a binding and from which that binding can later be retrieved. In ALAI, this function can easily be handled by tables - one simply indexes the table by a 'loc' pointer to the location which is to be saved. A pointer to the current context can be obtained from the safe fixed variable *MYCONTEXT. A similar variable, *ROOT, points to the context which is the root of the tree of contexts for the current world. 'Context' is a language defined record class with the following six fields:

*PARENT - points to the parent context of this context

*CHILDREN - a list of the children (contexts) of this context. A child is immediately above its parent in the context tree.

*BIRTHPLACE - the world which this context is part of

*LEVEL - the integer level in the context tree of this context. *ROOT is at level 0.

*CHANGES - this is the end node of a circular linked list containing the information necessary to switch between this context and its parent. Its format is complicated, hence the programmer is advised not to try to tamper with it.

*ROOTER - this is similar to *CHANGES, except that its information enables a direct switch between this context and *ROOT. Its format is not the same as that of *CHANGES, but the two are interwoven by necessity.

Through the *PARENT and *CHILDREN fields, contexts are arranged in a tree structure with *ROOT (it has no parent) as the root. The *BIRTHPLACE field is used to prevent attempts to switch to contexts which are parts of another world without also entering that world.

When a variable which has been declared 'back' or when a part of a variable which has been declared 'backall', etc. is assigned to, the fact of its value change is stored in the current context. The effect of such assignments on the *CHANGES and *ROOTER lists of descendent contexts is properly handled. Actual locations which are thus saved in one or more contexts do not contain their present value directly, instead they contain a pointer to a special record (actually a part of a *CHANGES list) which contains the current value. This seemingly inefficient method significantly increases the efficiency of contexts and their associated procedures. The procedure 'back' will dynamically save any location in such a manner. 'Unback' will remove all references to a given location from the entire tree of contexts. When a program block is exited, all locations in that block are automatically "unbacked". Because of this, contexts can be global objects even though they contain references to local values - all such references will be removed when the block in which the locals are declared is exited. Contexts are produced by the procedure 'sprout' and switched to (made the current context) by the procedure 'switch'. It is possible to reference values with respect to contexts other than the current one by postfixing the locations with 'wrt' followed by a context expression. The effect is to get the value that the location would have if that context were switched to. Assignments to a location 'wrt' a context have similar effects. Note that 'wrt' works with individual locations, hence '(A wrt C)(3)', 'A(3) wrt C' and '(A wrt C)(3) wrt C' may all be different.

The computation required to switch between contexts widely separated in the context tree can be prohibitive. To help alleviate this problem, the idea of "rooting" a context is introduced. "Rooting" a context requires the construction of a *ROOTER list for that context, which itself is a time consuming task. Switching between two rooted contexts, however, requires the traversal of only the two *ROOTER's rather than several *CHANGES lists. Thus if contexts which are widely separated in the context tree are to be switched into and out of often, it is wise to "root" them. Rooting and unrooting (to recover the space occupied by the *ROOTER list) are done by the procedures 'root' and 'unroot'. Sample program MISS_CAN2 (Appendix D) illustrates the use of contexts for saving "states of the world".

The following procedures are defined:

sprout - a child context is sprouted from the argument context and returned. The child's *PARENT is set to the argument context, its *BIRTHPLACE is set to *MYWORLD (the current world), its *LEVEL is set one greater that its parent's and its *CHILDREN, *CHANGES and *ROOTER are set to null.

switch - a switch is made to the argument context. The shortest path through the context tree (it may involve *ROOTERs) to the target context is found. This path is traversed by undoing all saved assignments when going down the tree (towards *ROOT) and re-doing all saved assignments when going back up. The effect is to give all locations the value they last had when the target context was the current context.

newcontext - sprouts a new context from the current context and switches to it. Effectively, 'newcontext' is 'switch sprout *MYCONTEXT'.

back - the location passed as argument is made backtrackable. Its current value is saved in *ROOT.

unback - all references in all contexts in the current world to the argument location are removed. 'Unback' can be applied to locations declared 'back', but its effect will be undone (to the extent that a new reference is produced) when an assignment to the 'back' location is made.

root - the context passed as argument is rooted, i.e. a *ROOTER list is made for it which condenses (and removes multiple assignments from) the *CHANGES lists traversed in switching from the argument context to *ROOT or vice versa.

unroot - the *ROOTER list of the argument context is destroyed

contextequal - returns true if the two context arguments are equivalent. Parentage, *ROOTER and *CHANGES are all checked.

4.29 Processes

A process in ALAI is something that can be executed as an independent entity. Procedures do not fall into this group because the execution of a procedure is a part of the execution of its caller, i.e. it is not independent. ALAI processes can be run serially or in parallel (or any mixture of the two) and can affect each other's operation if desired. Synchronization of and communication between processes is provided in the form of events (discussed in the next section). After its creation or incarnation and before its destruction, a process can be in one of four states: active, waiting, suspended or terminated. Active processes are currently "running", i.e. are on the active list of the scheduler (most computers do not support true parallel processing, so all active processes will not be executing but will take turns). Waiting processes are not currently active, but can be automatically made active when an event of the appropriate type occurs. Suspended processes must be explicitly "resumed" by an active process in order to become active. Terminated processes can never become active again but have not yet been destroyed. The safe fixed variable *MYPROCESS points to the currently executing process. 'Process' is a language defined record class with the following fields:

*BASES - this field is a list of stacks of areas. The areas are the storage regions used by various blocks for their local variables. The first element on the list is the stack for the current block (if it has one). Stacks are used to allow for recursive procedures and the calling of procedures which use block levels which are already in use.

*CONTINUE - this field indicates where to continue execution of a suspended or waiting process. It is either an 'expr' pointer to a part of program structure or a 'code' (a non user accessible data type) pointer into an actual region of machine code.

*SAVEAREA - this area is dependent on the machine and system of implementation. It is used for the storage of temporary values, e.g. register contents, relevant to the resumption of an inactive process.

*STATUS - this integer indicates the status of the process as follows: -2: terminated, -1: suspended, 0: active, >0: waiting. If the process is waiting, then *STATUS is one greater than the number of other processes waiting for the same event which will be triggered before this one.

*PATH - this field points to the end of the branch of the backtracking tree which this process is associated with.

*WAITFOR - this is the event_type which this process is currently waiting for (null if the process is not waiting)

*TRIGGERA - this is a list of the event_types which may be caused by the activation of this process

*TRIGGERI - this is a list of the event_types which may be caused by the inactivation of this process

*PRIORITY - this integer is the default priority (larger positive values mean higher priority) for this process when it is placed on a wait queue (waiting for an event)

*PROCONTEXT - this is the context which this process is currently working in (the value for *MYCONTEXT when this process is executing)

*PROWORLD - this is the world which this process is currently working in (the value for *MYWORLD when this process is executing)

*OBJECTS - a stack of elaborations which are currently being executed

*FIXLIST - a list of pairs containing a pointer to a location (type 'loc') and the value that location should be given when this process is resumed (see the procedures 'fix' and 'unfix')

*FINDLIST - this is a list of information needed to maintain continuity between equivalent 'find's

Processes are created by the 'incarnate' construct. The procedure given becomes the body of the process. Its execution will start at the start of the procedure and end at its end. When the end of such a procedure is reached, or the procedure 'return's, then the associated process is terminated. The integer expression following 'as' in the incarnate construct governs the interdependence of the incarnator (the process issuing the 'incarnate') and the incarnee (the process being created or incarnated). Safe fixed integers 'slave', 'independent', 'interdependent' and 'master', with values 1, 2, 3 and 4 respectively, represent the only values accepted. The identifiers used represent the status of the incarnee with respect to the incarnator. If the two are independent, then either can terminate without affecting the other; if the two are interdependent, then the termination of either will cause the termination of the other; the termination of the master will cause the termination of the slave; and the termination of the slave will not affect the master. Regardless of the relationship, when both are terminated, the notice of the termination is passed back up the (now mutual) execution path to any earlier incarnation point. This effect will become clearer when the execution path is explained in the discussion on backtracking. If a 'priority' expression is used, its value becomes the *PRIORITY of the new process, else the value of the fixed integer *DEFPRI is used.

When new processes are incarnated, their references to variables which exist at the time of the incarnation will normally be to the same copy as will references by the incarnator process. In some cases this is desired, but in others, the various processes should be independent with respect to what they do to program variables. This is a logical place to use contexts. To facilitate their use, a fixed boolean, *INCONT, is used to control the automatic sprouting of and switching to of new contexts on an incarnate. If *INCONT is true, then on incarnation, two new contexts are sprouted from the current one and the two processes each switch to one of the contexts. If *INCONT is false then no new contexts are created and the two processes both use the original context.

When a process is incarnated, it has status -1 (suspended). It can be made active by "resuming" it. The resume construct can be either a statement or an expression, the type being determined by how it is used. The integer 'time' expression (which defaults to zero) controls the nature of the resumption. If the value is zero then the subject process (the resumee) is immediately made active and execution of it replaces execution of the resumer which is suspended. (If parallel execution is in effect, then all of the time slots previously used by the resumer will now be used by the resumee.) If the 'time' value is non-zero then parallel execution commences (or continues if it was already in effect). Negative values assign that number (the absolute value) of consecutive time slots to the resumee. Positive values assign that number of time slots to the resumee, but the time slots are now evenly distributed in the scheduler's list of slots. If parallel execution was previously in effect, then the resumer's time slots are not changed for non-zero 'time' values, but if it was not, then the resumer is given only one time slot. The resumer continues execution after a paralleling resume until its time slot(s) is (are) used up for the moment. The resumer can be the resumee if the 'time' value is non-zero, in which case the number and distribution of its time slots are changed to yield the new arrangement. If parallel processing is not used then a 'resume' is essentially a switch to a coroutine.

The 'with' expression is any value to be passed to the process being resumed (the resumee) as the value of the 'resume' expression with which it had suspended itself. An attempt to pass a 'with' expression to a statement 'resume' or on the first 'resume' to a newly incarnated process will cause an error. Similarly, a process which suspended via an expression 'resume' cannot be reactivated by a 'resume' which does not pass a 'with' expression. It is an error to attempt to resume a process which is terminated, waiting or currently active. The 'wait' construct is used to make the present process wait for an event of the indicated type. The event_type is put as the *WAITFOR of the current process and the process is suspended and given a wait status dependent on the queue of processes waiting for an event of the given event_type. The process is put on the event_type's wait queue in a position determined by its *PRIORITY, which can be overidden by specifying a 'priority' expression. When an event of the appropriate type is caused (or if one was waiting to be noticed when the 'wait' was issued), and this process is next on the wait queue, then it is resumed and the value of the 'wait' expression is the event which caused the process's resumption. The process is resumed with one time slot alloted to it. When a process is incarnated its *BASES, *PROWORLD and *OBJECTS are copied from its incarnator; its *CONTINUE is the start of its body; *SAVEAREA is a new area of appropriate size; *STATUS is -1; *PATH is to the incarnation node just created; *WAITFOR, *TRIGGERA and *TRIGGERI are null; *PRIORITY is the priority expression or *DEFPRI; *PROCONTEXT is a new context or the same as its incarnator's; and *FIXLIST and *FINDLIST are null. Procedure PARAND (Appendix D) illustrates one use of processes.

The following procedures are defined:

terminate - the process supplied as argument is terminated. It is legal to terminate a terminated process.

suspend - the process supplied as argument is suspended. It is legal to suspend a suspended or waiting process, but not a terminated one.

schedule - the scheduler is called, i.e. the current time slot (or group of consecutive time slots assigned to the same process) is immediately ended

steal - the current process is requesting that it be given all time slots, i.e. that parallel processing be disabled. This is useful when a process is about to do manipulations which require that its environment, which it shares with other concurrent processes, not be tampered with until the manipulation is complete. For some computer operating systems, calling 'steal' before doing a large computation involving only localized values may tend to increase the overall efficiency of a program. The 'steal' request will be ignored (but will not be in error) if the current value of the fixed boolean *DENYSTEAL is true. Note that if a resume which causes parallel execution is given while 'steal' is in effect, the process(es) so resumed will not start their operation, i.e. will be given no time slots, until 'steal' is no longer in effect.

unsteal - returns to normal operation after a call to 'steal'

fix - the location passed as argument is added to the *FIXLIST of the current process. The values stored in the fixlist are updated when this process is resumed out of or is timed out of if parallel processing is in effect. When it is again activated, the locations indicated on its fixlist will be given the corresponding value from the list. The intent is to maintain independent values for fixed, language defined variables without the use of contexts, but other uses are also possible.

unfix - removes from the *FIXLIST of the current process any pair which saves the value of the location passed as argument

4.30 Events and Event_types

Events and event_types were mentioned in the discussion on processes. They are used to synchronize the operation of various processes and for the passing of information among them. Events also happen in the real world and in many model worlds which are simulated by computer programs. Because of this, one would like to be able to use some form of representation for events in the data net. ALAI events are valid data net components and, unlike most other data types, events contain a field, *KNOW, which makes the net knowledge about an event accessible from that event. Thus ALAI events are both valid semantic entities for use in the data net and operational entities which can directly affect program execution. 'Event_type' is a language defined record class with the following fields:

*WAIT - a queue of the processes waiting for an event of this type

*NOTICE - a queue of events of this type waiting to be noticed. Normally, one or both of *WAIT and *NOTICE will be empty.

*WHAT - any arbitrary value describing this event_type. In the case of compound event_types (built up from one or more processes) using the constructs 'active', 'inactive', 'any' and 'all', *WHAT will be a special purpose descriptor.

*KNOW - this is a list of all of the instances in which this event_type has been used as a participant

'Event' is a language defined record class with the following fields:

*MESSAGE - this is any arbitrary value associated with this event. It will usually be a particular description which distinguishes this event from others of its kind.

*TYPE - this is a pointer to the event_type which this event is a representative of

*KNOW - this is a list of all of the instances in which this event has been used as a participant

The 'any'/'all' - 'active'/'inactive' construct allows for the definition of event_types from processes. If only one process expression is used, then the 'any'/'all' is omitted and an event of this type is caused by the activation ('active') or the inactivation ('inactive') of the specified process. The extension to multiple processes and 'any'/'all' is straightforward, e.g. the event_type 'any active (P1,P2,P3)' is "caused" when any of the processes P1, P2, P3 become active (it must have been previously inactive). 'All' means the moment when all of the processes become active or inactive (actually, the last one of the group to become active or inactive while the others are all in the appropriate state will cause the event). If an event_type is defined in such a way, then the *WHAT field of it will be used to indicate the processes involved. For one process only, *WHAT will be an 'expr' whose *REL is *ACTIVE or *INACTIVE and whose single participant is the process involved. For 'any'/'all' cases, *WHAT will be an 'expr' with *ANYACTIVE, *ANYINACTIVE, *ALLACTIVE or *ALLINACTIVE as *REL and the processes as participants. When such an event occurs, its *MESSAGE will be the process which immediately led to its occurrence, i.e. the last one to go into the desired state.

Because actual processes do not exist until run time (i.e. there are no process constants), event_types which depend on processes cannot be constructed until all of the relevant processes are in existence. To conform to this requirement, the 'any'/'all' - 'active'/'inactive' construct is executable, i.e. it is executed every time it is encountered. Thus the user should avoid executing the same or equivalent event_type specifications more than once, since this would lead to redundancy of event_types. If one or more of the processes changes, however, it is necessary to construct a new event_type. Event_types which will no longer be used should be destroyed.

The 'cause' statement can be used to cause an event of the specified type. The 'with' part is used to supply a value for *MESSAGE of the new event; if it is omitted, then the current value of the fixed integer *EVENTNO is used and *EVENTNO is incremented by one. The 'on' statement is used to set up a trap for an event_type. The 'on' trap in effect for a given event_type is the one which was executed most recently. The traps are pushed onto a stack by 'on' and can be popped by the procedure 'popon'. If such a trap is in effect, then when an event is caused, it is placed on the notice queue (regardless of the state of the wait queue) and the block in the 'on' statement is executed. The block is free to do whatever it wishes to the event and the queues. If, when the block finishes (or calls the procedure 'continue'), both the wait queue and the notice queue are not empty, then events and processes are successively removed and paired up until one queue is empty. The processes so removed are started and the event they were paired with is the value for the 'wait' expression which put them on the wait queue. At the same time, the process(es) running at the time of the event are restarted (they were all temporarily stopped when the event was trapped). The safe fixed variable *INTERRUPT is an event_type used for various kinds of hardware, software and user interrupts. The nature and handling of the events of this type will be dependent on machine, environment and programmer of the particular implementation. Standardization is possible, but has not been looked at. Sample procedure PARAND (Appendix D) illustrates one use of events.

The following procedures are defined:

popon - pops the last 'on' trap from the system maintained stack of traps for the argument event_type. If no trap is left on the stack, then trapping is disabled.

continue - exits from the body of an 'on' trap and restores normal execution (possibly with the addition of processes which were waiting for the event which was trapped).

newevent - a new event_type with description as passed as argument is created and returned

eventequal - returns true if the two event_type arguments are equivalent

4.31 Backtracking

Backtracking is the process of executing a program backwards in such a way that variable values are downdated, i.e. given the values they had earlier. In AI work backtracking is usually used to try several different methods or assumptions for solving some kind of problem. When one method fails to solve the problem, the problem solving program is backed up to undo all of the things done in trying the method and another method is then tried. If a method or assumption does not result in a failure, then it is not backtracked and its effects are carried over to later parts of the program. In ALAI, explicit failures cause backtracking to specific backtrack points which are dynamically produced by an executing program. These backtrack points are nodes in the backtrack tree, which is essentially a tree representing the course of execution of the current processes. It has one branch for each process currently in existence. The links in the tree point backwards, i.e. towards the root. The nodes in the tree are of four types: backtrack points, process incarnation points (at these points a branch splits into two branches), block entry points for blocks in which backtrack points or incarnation points have been produced (there will be a block entry point for all current blocks, regardless of whether or not they are needed) and method entry points (discussed under 'Elaborations').

Incarnation points are produced by the execution of the 'incarnate' construct. They are added at the end of the current backtrack tree as pointed to by *PATH of the process issuing the incarnate. The *PATH of the two resulting processes will then point to the new node which consists of pointers to the two processes, an integer indicating the status of the right-hand process (the second field of the node) with respect to the left-hand process (the first field of the node) and a pointer to the previous node in the tree. The status code is exactly as in the incarnate 'as' expression. For execution purposes, blocks are considered to be a sequence of statements, perhaps followed by an expression, which has an associated set of declarations. Syntactic blocks with no declarations are not considered to be blocks, they are simply a sequence of statements. When a block (with declarations) is entered at execution time, a block entry node is produced and added to the current branch of the backtrack tree. It consists of an integer block level number (levels for blocks with declarations) and a pointer to the previous node. When such a block is exited, if its entry node is the last node in the tree, then it is deleted since there is no way to back up into the block and there is no other process dependent on its continued existance. At the same time, the storage assigned to the variables in the block is released.

Backtrack points are produced by executing the 'try' and 'ptry' constructs, both of which are essentially special purpose interative constructs. The optional strings at the end of them are used to label the backtrack points produced by them so that backing up can be more closely controlled. The body of the 'try' statement, corresponding to the block used as a body for the iteration statement, is whatever program is executed after the 'try', including itself if loops, iteration or recursion are involved. For example, consider the following 'try' statement:

```
   for X via GEN for I by 1 while GOOD(X) try "netsearch"
```

When encountered during forward execution, this statement will set up a backtrack point labelled "netsearch" pointing to the 'try' statement. GEN will be called to produce a value for X, I will be initialized to 1, and then, if GOOD(X) yields true, execution continues with the program following the 'try' statement. If, during this program, a fail to "netsearch" occurs, the iteration steps will be performed again, i.e. GEN will be called to yield a new value for X, I will be incremented by 1 and GOOD(X) will be called. The program following the 'try' will then be executed again with the new values for X and I, along with any effects of the calls to GEN and GOOD, until a fail to "netsearch" occurs. This cycle is repeated until the program after the 'try' "succeeds" in which case the backtrack point is removed, or until GEN yields null or GOOD(X) yields false, in which case the 'try' statement fails and further backup occurs.

The 'ptry' expression is of a similar nature, but it offers greater control over the trying of the alternatives. Consider the following:

```
P:=*MYPROCESS;
L:=for X via GEN while GOOD(X)
   ptry
   BLOCK1;
   resume P;
   BLOCK2
   then;
```

BLOCK1 and BLOCK2 represent sections of the ptry-then block which do the actual processing. On each step through the iteration, GEN is called to produce a value for X, GOOD is called to test that value, a process with the ptry-then block as body is incarnated as slave of the current process and the new process is resumed, replacing the current process. BLOCK1 is executed and then the new process resumes P which then places the new process in the list whose location is given as value to L (this assignment is done before any processes are incarnated). P, the main process, then iterates again, producing another new process with the same body as before. This sequence of actions continues until GEN returns null or GOOD(X) returns false. The 'ptry' is then removed from the backtrack tree and all that is left is L which contains a loc pointing to a list of processes, all containing the same block and all suspended at the same point. Each of these, however, has a (probably) unique value for X. The program after the 'ptry' can then supervise the running of these processes in any way it sees fit. If any of the processes fails or runs into the end of BLOCK2, then it is terminated and cannot be resumed. If a fail occurs when processing BLOCK1, or if there is no resume to get the next process, then the new process is terminated and is not included in the list of processes which L points at (through one step of indirection). The iteration variables (X in the example) are all redeclared in the ptry-then block and initialized to the values of the corresponding original iteration variables. This allows each process to have separate and independent values without the need of contexts. A new context will be sprouted and switched to for each new process only if the fixed boolean *PBTCONT is true.

Similarly, new contexts will be sprouted and switched to for each iteration in a 'try' only if the fixed boolean *BTCONT is true. The sample procedure 'PARAND' illustrates the use of 'ptry'.

Backtrack points are produced by 'try' and 'failing' and are temporarily used by 'ptry' and 'attempt'. They consist of a string label (the string used at the end of the 'try', 'ptry' or 'failing' which produced the point), a pointer to the program structure or machine code to be executed after backing up to this point, a context (may be null, in which case no context switch occurs) to switch to before continuing execution, an integer indicating how many *BASES stacks to pop in order to get back to this point, and a pointer to the previous node in the tree. If the backtrack point was created without specifying a string as label, then that field of the node will be empty (null). The 'failing' construct is used as a trap for backtracking; when encountered during normal execution, it is skipped, but when backed into, it will be executed. When it has executed, program flow will continue in the forward direction (with the 'failing' reactivated) unless the body of the 'failing' generated a fail of its own.

The 'attempt' statement is essentially a 'cond' statement with success and failure playing the roles of true and false. If the first part of a pair (the two parts are those separated by a colon) succeeds, i.e. execution drops out of the bottom of it, then the second part is executed. No more than one second part is ever executed, but all of the first parts may be attempted and fail. If no first part succeeds, then the 'else' part, if present, is executed. The 'attempt' construct may be used, by letting the second part merely set a flag, as a test for success of any or all of several alternatives, i.e. as an AND or OR of successes.

Three procedures are used to initiate backtracking (it is also initiated when a 'try' runs out of alternatives): 'fail', 'failto' and 'failpast'. 'Fail' fails back to the latest backtrack point on the tree, 'failto' fails back to the last backtrack point whose label matches the string passed as argument and 'failpast' fails to the point immediately before the latest one whose label matches 'failpast's argument. Backtrack points failed over are removed from the tree. Normal forward execution continues in the appropriate place in the construct which produced the backtrack point backed up to. The point is maintained as active in the tree unless the construct generates a fail of its own ('failing' has a fail in it or 'try' runs out of alternatives) in which case it is removed and a simple unlabelled fail to the next point occurs.

Backtrack points can be removed from the tree by direct manipulation or by the procedures 'succeed', 'succeedto' and 'succeedpast'. 'Succeed' removes only the latest backtrack point, 'succeedto' removes all points up to but not including the latest one whose label matches 'succeedto's argument and 'succeedpast' removes all points up to and including the latest one whose label matches the argument. These procedures are used for finalizing the choice of alternatives and for cleaning up backtrack points that are no longer useable. The 'fresume' construct is the same as the 'resume' construct (with no 'with' allowed) except that a fail is immediately generated in the procedure being resumed. With proper use of 'failing' and 'fresume', notice of failure can be backed up across process changes.

When more than one process is in use, it is necessary that the various *BASES lists be separate in order that a block entry or exit in one process will not affect the other processes. Similarly, it is desirable that block entry and exit not affect the generality of backtracking; specifically, it is desirable to be able to back into a block which has been exited and have the local variables in that block become available again with their latest values. The ALAI backtracking methodology allows this without the need of contexts. When a backtrack point is produced, the top elements of the active stacks on the current *BASES list are duplicated; one copy is a working copy which can be changed as blocks are entered and exited, but the other copy is never changed until the backtrack point is removed and thus can always be used to restore local variables. When a block (one with local variables) is exited, if the latest node on the branch of the backtrack tree is not its entry node, then the storage for the local variables in that block is not freed. The top of the *BASES stack which points to it can be changed by a later block entry without "losing" the locals since it will have been either duplicated or copied in the *BASES list of another process (one which was incarnated in that block and hence is dependent on it).

The four types of nodes used in the backtrack tree cause several different types of actions when encountered during backup. Backup through a block entry point indicates that everything (if anything) that was dependent on the local variables of that block is now gone, hence the action taken is to remove the node and free the storage occupied by that block's locals. Backtrack points which are backed up over are simply removed from the tree. The appropriate number of stacks on *BASES are popped once and the indicated context switched to. Backtrack points which are backed up to are first backed up over without removing the node from the tree, i.e. stacks are popped and a context switch may occur. Further execution may then re-establish the backtrack point or cause a further failure. Action taken on reaching an incarnation point depends on the status of the other process and the dependency relation between the two. If the failing process is the slave or the two are interdependent and the other process is not yet terminated then the process is terminated; if the failing process is the master or the two are interdependent, then both processes are terminated and destroyed, the node is removed and backup continues; else if the processes are independent and the other is already terminated, then the failing process is terminated, both processes are destroyed, the node is removed and backup continues.

Several global (fixed) switches are associated with backtracking: a new context will be sprouted and switched to for each process formed in a 'ptry' if and only if *PBTCONT is true; a new context will be sprouted and switched to for each new alternative in a 'try' if and only if *BTCONT is true; assignments to the data net will be saved in contexts if and only if *NETSAVE is true; assignments to all parts of all program variables regardless of their 'back' status will be saved in contexts only if *ALLBACK is true; and the saving of any values at all in contexts is disabled if *ANYBACK is false. Note that process switches change to new branches of the backtrack tree which will have different nodes than the old one. Contexts are not automatically destroyed on backtracking (the user may have saved them somewhere in order that he can look to see what a 'try' had accomplished) so the user is responsible for destroying those he no longer needs. The garbage collector will not destroy any contexts since they are all linked together via *PARENT and *CHILDREN links. The sample programs MISS_CAN1 and MISS_CAN2 (Appendix D) illustrate the use of backtracking. One use of 'ptry' is shown in the sample procedure PARAND.

The following procedures are defined:

fail - fails to the most recent backtrack point on the current branch of the backtrack tree

failto - fails back to the most recent backtrack point on the current branch of the backtrack tree whose label equals the string passed as argument to failto

failpast - fails back to the backtrack point created just before the most recent one on the current branch of the backtrack tree whose label equals the string passed as argument to failpast

succeed - removes the most recent backtrack point from the current branch of the backtrack tree

succeedto - removes all backtrack points on the current branch of the backtrack tree up to but not including the most recent one whose label equals the string passed as argument to succeedto

succeedpast - removes all backtrack points on the current branch of the backtrack tree up to and including the most recent one whose label equals the string passed as argument to succeedpast. i.e. 'succeedpast("point"); fail' is equivalent to 'failpast("point")'.

4.32 Worlds

Contexts are usually used to represent a state of the world or of some system within which a program can be said to be operating. The form used in ALAI is especially suited to contexts which do not vary much from parent to child. The few differences are nicely contained in the list of changes. If, however, the user wishes to work in environments which are drastically dissimilar, the context mechanism becomes highly inefficient, in terms of both storage needed and computation used in switching contexts. This could easily happen if, say, a generalized problem solver is working in a world of toy blocks and a mechanical arm, and then must switch to a world of predicate calculus. The two problem domains have structures in common, but these would be far outweighed by the differences. The situation becomes almost impractical if the problem solver is to solve problems in both domains simultaneously - it would then be continually switching between domains and would spend most of its time doing so. ALAI offers a construct which combats this problem. 'World' is an ALAI data type, denoting objects which contain entire worlds - i.e. a data net with associated dictionary (this automatically includes a tree of contexts). A program can create as many worlds as it wishes and can enter them at will. Interworld referencing for data net search and assertion is provided. Each process has associated with it a world of operation and a current context within that world, both of which can be changed at will.

There are no world constants in ALAI, hence all worlds must be constructed dynamically. This is done by the procedure 'neworld'. The current process can change its world of execution (pointed to by the safe fixed variable *MYWORLD) by using the 'enter' construct. 'Enter' causes a switch to the specified world with conditions as contained in the indicated context. If the 'wrt' part is omitted, then the world is entered with the same context it was in when last exited (by an 'enter' out of it). If all operations in a world have been with *NETBACK turned off (false), then the context used is irrelevant to the world itself, but may still be significant as far as program variables are concerned. The 'assert' expression will construct sufficient instances under the specified context in the specified world to make a structure in the net as described by the description given. A pointer to the top level instance of the resulting structure is returned. The context defaults to the last-used context of the world, which in turn defaults to the current world (*MYWORLD). The 'delete' statement, used for removing things from the data net, has two forms. If an instance is used, then that specific instance is deleted from the net (in whatever world it is in) with respect to the context specified. The deletion is effective only in the context specified and all descendents of it which have not altered a part of it. If a descriptor is used, then an instance corresponding to that descriptor, in the world specified, is found (pattern matching is not used, the correspondence must be exact) and deleted with respect to the specified context. The world and context default as in 'assert'.

The following procedure is defined:

neworld - creates and returns a new world. The world is empty except for a few system concepts.

Several different applications are possible for worlds. One, mentioned above is that of different problem domains for a problem solver. Another use for worlds is as containers for models, i.e. for small subworlds which represent some part of a larger world, but in a form which is easier to handle. Other uses come from attempts to handle full natural language. When the words "suppose that" occur in English discourse, we usually expect the next few words or sentences to describe a situation which is probably not valid, but, for the sake of discussion, we are to assume that it is. ALAI worlds can represent such suppositional worlds in which "facts" may be directly contrary to knowledge in some other world. Human belief systems, which are effectively individuals' personal models of the real world, are also amenable to representation as ALAI worlds. The following sample program skeleton, which is basically similar to that of the example at the end of section 4.27, illustrates this latter use of worlds.

```
D FACT;
/* the description representing the fact to  be  asserted
or looked up */
I PERSON;
/*  the  net structure for the person who's belief system
we are in */
world table WORLDOF;
/* table of the various worlds indexed by the PERSON */
string INPUT;
bool QUESTION, DELETE;
world MAIN:=*MYWORLD;
bool proc ENDER(string X);
/* tests for a request to stop */
proc PARSE(string X);
/* Parse the input English string, set QUESTION if  is  a
question,  DELETE  if  is a request to delete a fact. Set
PERSON (creating new  concepts,  instance  and  world  if
necessary) to the instance representing the one doing the
believing,  or  to  NULL  if  no  believer. Set FACT to a
description representing the fact(s) to  be  asserted, or
searched for. */
/* commence actual processing */
WORLDOF(null):=*MYWORLD;
while INPUT:=getcard; ¬ENDER(INPUT)
do PARSE(INPUT);
   if ¬QUESTION&¬DELETE
   then assert FACT in WORLDOF(PERSON);
   elif ¬DELETE
```

```
        then enter WORLDOF(PERSON);
           /* answer the question if possible */
           enter MAIN;
           /* if not yet answered, then answer it now */
        else delete FACT in WORLDOF(PERSON)
        fi
    od
```

4.33 Elaborations

The idea of an elaboration (type 'elab') is being introduced in ALAI as an experiment. They may prove useful or they may prove completely useless. The intent is to introduce further linkage between semantic structures stored in the data net and executable program structures. They also give a new control structure which can be viewed as hierarchical in nature and which can complement the facilities offered by the 'try' statement. 'Elab' is a language defined record class with the following fields:

*OBJECT - this is the net structure (node) which is the semantic meaning or intent of the elaboration, i.e. the structure which is being elaborated

*METHODS - this is a list of expressions or pointers to executable machine code. These methods are the routines which attempt to elaborate or perform *OBJECT.

*ELABNO - this is an integer index into *METHODS which indicates the "current" method, or, if it is one greater than the length of *METHODS, it indicates that a new method is to be produced before this elaboration can be executed

*KNOW - this is a list of all of the instances in which this elaboration has participated

Elaborations are valid statements and as such they appear in executable programs as part of the program itself.

When elaboration constants are defined (the identifier before the '!' is the name of the constant), they are given the node expression as *OBJECT, and the list of 'expr's (else null) as *METHODS; *ELABNO is set to 1 and *KNOW to null. The evaluation of the relevant expressions is done on entry to the block in which the elaboration constant appears. The 'local'/'global' special modifier is relevant for 'elab' variables: local variables can accept any 'elab' value, but their value cannot be passed out of their block of declaration; global values can be passed anywhere, but their *METHODS lists are restricted to 'expr's which are fully global, i.e. reference only net and fixed variables. The globality of an 'elab' constant is determined by the globality of the most local variable referenced in the 'expr's given.

A fixed 'procval' variable, *ELABORATOR, is intimately associated with the use of elaborations. When an elaboration is encountered during execution, its current (its *METHODS indexed by its *ELABNO) method is accessed. If the method exists then this elaboration is pushed onto *OBJECTS of the current process and the method is executed. At the same time a method entry point is added to the current branch of the backtrack tree. The point consists of only a pointer to the previous node in the tree and it serves only to insure that *OBJECTS is popped appropriately on backing up. For all other language-defined uses of the tree it is ignored. If the method being executed completes its execution, control returns to the point following the elaboration, *OBJECTS is popped and the method entry point is removed. If accessing the current method yields a null pointer, then the current value of *ELABORATOR is called with the elaboration as argument. The intent is that the elaboration procedure produce a method to be used or change *ELABNO to indicate an existing method. If, on return from the elaborator, the current method is still null, a fail is generated, indicating that no alternative was found. If the elaborator did yield a current method, then that method is executed in the manner described above. The hierarchical nature of elaborations comes from the fact that a program being executed as a method can itself contain elaborations. Also, any program can access *OBJECTS of the current process to find out what elaboration(s) it is a part of.

The expected uses of elaborations are such that a large surrounding program capable of problem solving, deductive reasoning, etc., is needed. Hence a true example of their intent is not possible here. Instead, a verbal "example" is given. Suppose that the reasoning system has been asked to produce a plan for performing some task, e.g. build a two storey house out of wooden blocks. Initial task analysis might produce a sequence of substeps, stored in the semantic net, such as:

```
(BUILD,LARGE_BASE)
(BUILD,FIRST_STOREY)
(BUILD,SECOND_STOREY)
(BUILD,ROOF)
```

Such a plan must be interpreted by a plan follower in order to be carried out. The ALAI elaboration feature allows the plan to be directly converted into an executable program segment:

```
BASE!NI(BUILD,LARGE_BASE);
FIRST!NI(BUILD,FIRST_STOREY);
SECOND!NI(BUILD,SECOND_STOREY);
ROOF!NI(BUILD,ROOF);
```

This program segment (after setting *ELABORATOR to a routine which can accomplish the various parts, however inefficiently) can then be treated as an unevaluated expression and directly executed.

So far, little has been gained. As more is learned, however, and as the segment is executed repeatedly, the resultant details of its operation will become clearer. As patterns arise, program segments for the subtasks can be produced and assigned as methods of the top-level elaborations. These sub-task routines could also consist of elaborations and basic steps. The cycle of learn details, fill in methods, etc. can be repeated until the various steps become trivial and directly programmable. The result will be a fairly efficient program which accomplishes the task directly and which is "commented" by the *OBJECTS of the various elaborations which embody it. Alternate solutions found would be stored as other *METHODS and are thus easily available for comparison purposes. The final step of producing a normal program would only require the concatenation of all of the various bottom level steps. The advantages of this methodology are its uniformity and generality: a routine can be directly executed whether it is fully detailed, just created in rough form, or in some intermediate developmental stage, and the elaboration scheme allows different parts of the routines to be in very different states of development. The automatic existence of semantic comments (*OBJECTS) makes the resulting routine comprehensible to semantic routines at all but the ultimate stage of development.

4.34 Odds and Ends

The fixed boolean *NETSORT controls the application of a heuristic intended to speed up data net searches. If *NETSORT is true, then each reference to an element of a participation or usage list which was used for further searching will cause that element to be interchanged on the list with the element before it. The intent is to "sort" the lists so that the most used references on them will be towards the front of the list. This action is disabled if *NETSORT is false. The fixed boolean *PERMIT controls the use of the fields of language defined record classes and of safe variables as receptors in assignments and as arguments for the '@' 'loc'-generating operator. If *PERMIT is false when such a use is parsed, the usage will be disallowed. *PERMIT should be kept false unless it is absolutely necessary to change some portion of a language defined entity. The intent is to prevent the user from "fouling up the works" by tampering with values which the language is dependent on. Certain additional procedures are needed for the language but have not yet been described; they are described here.

The following procedures are defined:

copy - will yield a copy of its string, pair, triple, vector, set, bag, list, chain, stack, queue, record, array, table or descriptor argument. The parser is aware of 'copy' and will use an appropriate type for it.

size - integer result is as follows:
1) if argument is string, then the length of the string
2) if argument is pair, then 2
3) if argument is triple, then 3
4) if argument is vector, set, bag, list, chain, stack or queue then the number of elements in the argument
5) if argument is array, then the total number of elements in the array
6) if argument is record, then the number of fields
7) if argument is descriptor, then the number of participants+1
8) if argument is table, then the number of entries
9) if argument is elaboration, then the number of methods
10) if argument is area, then the size of the area
11) else 1

ignore - this procedure accepts a single argument of any type. It ignores that argument, does nothing and returns no result. It is used to turn an expression into a statement.

destroy - this procedure accepts a pointer to any record-type entity (including language defined records) and destroys it, i.e. returns the space it occupies to the storage manipulator. 'Destroy' does not do any special safeguarding to prevent other references (now invalid) to the thing destroyed, but it will undo back links from participants to an instance or descriptor being destroyed.

typeit - this procedure returns its second argument modified to have the type passed as first argument (integer)

getlin - the string argument is decoded into a linear sequence, e.g. a list, set, instance, etc. and a pointer to the decoded result is returned. The expected format in the string is a typing pseudofunction, '<', a list of constants and a closing '>'. Other 'get...' procedures will be used for processing the constants and the fixed boolean *LINERR will be set to true if the format of the linear sequence is bad.

getseq - this procedure is like 'getlin' except that the '(...)' format is expected, complete with the convention of the carrying over of the main type to internal subsequences. The error flag is *SEQERR.

replin - this procedure is the reverse of 'getlin' - it accepts a linear sequence as argument and returns a string representation using the double angle bracket format

repseq - this procedure corresponds to 'getseq' as 'replin' corresponds to 'getlin'

CHAPTER 5

CONCLUSION

This thesis has discussed the programming language needs of artificial intelligence workers. Four specific languages, LISP, SAIL, SNOBOL and 2.PAK were discussed with respect to the features they offer which are useful to AI workers. It was found that no language offers all of the features that would be needed and thus that no language would provide a practical base for a general artificial intelligence system. It was decided that a new language, embodying most of the desirable features, would be a useful tool for AI research. Such a language, ALAI, was proposed and described.

ALAI is based on ALGOL and contains basic facilities such as arithmetic, character manipulation, arrays, etc. which are provided by most standard programming languages. This base is augmented by many data types more specialized in their usage, e.g. sets, bags, tables, stacks and queues. A built in data net provides facilities for the easy storage of semantic knowledge. The net can be searched via pattern matching techniques embedded in the powerful iteration facility. Similar pattern matching, as concise and powerful as that of SNOBOL or SAIL, can be applied to arbitrary data, including character strings. Failure driven backtracking, again based on the iteration construct, allows the incorporation of trial and error searches into normal programs. More controlled search is provided by the 'ptry' statement form which generates lists of selected processes, the elements of which can be individually controlled to evaluate or try a given possibility. The techniques and needs of multiple processes, backtracking and data net operations have been successfully combined in a consistent and usable system of conventions.

The capability for dynamic program modification and generation is supplied by defining the internal workings of the system to be an integral part of the language. Programs are represented as temporary parts of the data net and can be manipulated by the pattern matching facility. Automatic program production is facilitated by a new data type, elaboration, which bridges the conceptual gap between pure semantic structure and pure program structure. The inherent inefficiencies associated with free use of contexts and the data net have been reduced by the use of usage and participation counts on all concepts and by the introduction of rooted contexts and of worlds. It is not expected that users would wish to manipulate the internal form of such contexts even though the capability is provided.

A working version of ALAI has not yet been implemented. Work is soon to begin on an initial version, however, and it is hoped that a working interpreter (and perhaps compiler) will soon be produced. A proposed external appearance of the ALAI system is discussed in Appendix C. The initial implementation will start with a set of basic procedures written in assembly language (e.g. iplus, itimes, sconc, code, logand, etc.) and routines, again written in assembly language, to interpret the lowest level operations of the language (e.g. simple assignment, identifier location, basic 'if' statement). Also needed is a minimal interface to the outer world, so that communication is possible. These components represent the base portion which would have to be coded by any installation wishing to implement the language.

Following these initial steps, higher level procedures and higher level interpretation routines would be written. This next level would be entirely interpretable by the lowest level assembly language routines and would require no procedures other than those existing at the lowest level (or others at the same level but already defined). In the initial implementation only, a routine would be needed to translate from an external form (probably similar to basic LISP because of its syntactic simplicity) to internal structures which can then be interpreted. This process of building up on lower levels can continue until all language components, i.e. interpretation routines, predefined procedures, parser, compiler, editor, etc. are complete.

This paper has, hopefully, drawn greater attention to the pressing need for standardization in AI work. This standardization is most easily obtained through a single programming language which is easily portable and for which one standard implementation exists. The ALAI language definition shows that the features and capabilities needed for the various types of AI programming can be combined in a single language which is still usable. The presence of the various features which would not normally be needed for a given task can make the resultant program more general and more pleasing in its operation, as exampled by the free form input of the procedure 'NIM' in Appendix D. Such a combination of powers may also free the mind of the programmer from many of the details of his task, thus leaving more time and patience for the development of better, more powerful programs for artificial intelligence.

Although ALAI is powerful and general, it is definitely not the ultimate AI language. The basic style, that of an ALGOL base with special "reserved words" used for syntactic constructs representing the semantic capabilities of the language, may be replaced by a scheme of high density like that of APL, or of high verbosity like that of COBOL, or, more likely, by a form as yet unanticipated. The selection of features for the language was based on past usefulness and guesses as to future need, hence may prove to be completely inappropriate. New capabilities and techniques will likely be introduced which will supercede or make obsolete many of the ones in ALAI and will provide capabilities not offered by ALAI. However, it is hoped that actual use of ALAI will produce working AI programs of greater power and usefulness than has previously been obtained. The experience thus gained will undoubtedly aid in the discovery of new programming techniques and new ways of thinking about old problems.

The original impetus for the design of ALAI came from the realization that none of the readily available AI languages suited the needs of a planned natural language comprehension/reasoning system which could be easily transported. Since then, the design and implementation of the language has become a major goal in itself, but because the original goal was not lost, the design of the language may reflect the desire to simplify the programming of natural language and reasoning applications. Regardless of its many shortcomings, it is felt that the language will be a definite aid to AI projects, at least locally. The ultimate programming language - a verbal natural language backed by comprehensive planning, deduction, induction and automatic programming facilities - may thus be closer because of the successful design of a language more suited to the needs of those who are to implement the ultimate language.

References

Bobrow, D. G. and B. Raphael, "New Programming Languages for Artificial Intelligence Research", *ACM Computing Surveys*, *6*, No.6, Sept. 1974.

Burstall, R. M. and R. J. Popplestone, "POP-2 Reference Manual", *Machine Intelligence 2*, E. Dale and D. Michie eds., American Elsevier, 1968.

Griswold, R. E., J. F. Poage and I. P. Polonsky, *The SNOBOL4 Programming Language*, Prentice-Hall, 1971.

Hewitt, C., "Planner: A Language for Proving Theorems in Robots", *Proceedings of the International Joint Conference on Artificial Intelligence*, 1969.

Hewitt, C., "Procedural Embedding of Knowledge in Planner", *Proceedings of the Second International Joint Conference on Artificial Intelligence*, 1971.

McDermott, D. V. and G. J. Sussman, "The Conniver Reference Manual", *AI Memo #259*, MIT Artificial Intelligence Laboratory, May 1972.

Melli, L. F., "The 2.PAK Language Primitives for AI Applications", *Technical Report #73*, University of Toronto, December 1974.

Rulifson, J. F., J. A. Derksen and R. J. Waldinger, "QA4: A Procedural Calculus for Intuitive Reasoning", *Technical Note 73*, Artificial Intelligence Center, Stanford Research Institute Project 8721, November 1972.

Smith, D. C., "MLISP", *Memo AIM-135*, Stanford AI Project, October 1970.

Sussman, G. J. and D. V. McDermott, "Why Conniving is Better Than Planning", *AI Memo #255A*, MIT Artificial Intelligence Laboratory, April 1972.

Van Lehn, K. A. ed., "Sail User Manual", *Memo AIM-204*, Stanford AI Lab, July 1973.

Van Wijngaarden, A., B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens and R. Fisher, "Revised Report on the Algorithmic Language ALGOL68", *Acta Informatica*, 1976, *5*, in press.

Weissman, C., *LISP 1.5 Primer*, Dickenson, 1967.

Appendix A - A Grammar for ALAI

The parser for ALAI detects type mismatches, multiple or nonexistent declarations, invalid parameter lists, etc., all of which are semantic errors rather than syntactic errors. Complex grammars can include such things as type checking, operator precedence, etc. in the syntax of the language, but the following grammar does not do so. Instead, it merely states the form of the ALAI constructs, rather than the way in which they should be used. The meta-syntax used to describe the ALAI syntax is a modified form of a B.N.F. grammar. Meta-symbols, sequences of letters and hyphens enclosed in angle brackets ('<' and '>'), are used as non-terminal symbols indicating concepts which exist in some form in the ALAI syntax, but which are more complicated than simple strings. Examples are <integer>, <declaration> and <expression>. The form of the meta-symbols is specified by productions of the form:

meta-symbol ::= meta-expression

Meta-expressions are built up of simple constants (single or sequences of characters which stand for themselves) and various elements of meta-syntax:

1) {meta-expression-1 | meta-expression-2 | ---
| meta-expression-n}

The resulting meta-expression can be any of the sub-meta-expressions. If a meta-expression of this type is the only meta-expression in a meta-symbol specification, then the enclosing brace brackets are omitted.

2) meta-expression-1 meta-expression-2 ---
meta-expression-n

The resulting meta-expression is that constructed by linearly concatenating or appending all of the sub-meta-expressions.

3) [meta-expression]

The bracketed meta-expression is optional, i.e. it need not appear.

4) ≤meta-expression-1 | meta-expression-2 | ---
| meta-expression-n≥

At least one of the sub-meta-expressions must appear. If more than one appears, then they must be in the same order as they appear in the list of alternates.

Upper case letters appearing in meta-symbols stand for variables and must be substituted for consistently throughout individual productions. Such a 'T', i.e. as in <T-expression> stands for any ALAI type as represented by the meta-symbol <type-1>. In the actual parser intput, blanks are ignored except as parts of strings. They do serve as component delimiters, however; 'event_ type' is not equivalent to 'event_type', nor 'PERSON A' to 'PERSONA'.

Identifiers

```
<letter> ::= A | B | C | D | E | F | G | H | I | J | K |
   L | M | N | O | P | Q | R | S | T | U | V | W | X | Y
   | Z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<identifier> ::= <letter> | <identifier> {<letter> |
   <digit> | _}
<any-identifier> ::= [*] <identifier>
```

Declarations

```
<user-type> ::= <identifier>
<simple-type> ::= int | real | bits | bool | string |
   label | dict | C | I | N | D | CD | ID | ND | pat |
   procval | process | context | event | event_type |
   world | elab | area | free | any | <user-type>
<compound-type> ::= pair | triple | vector | set | bag |
   list | chain | table | array | stack | queue | expr
<type-1> ::= <simple-type> | <compound-type> |
   ref[ (<identifier>) ] | <type-1> <compound-type> |
   [ {<type-1> | ptr | fixed} ] loc
<number> ::= <digit> | <number> <digit>
<integer> ::= [ {+ | -} ] <number>
<bound-pairs-1> ::= [<integer> :] <integer> |
   <bound-pairs-1> , [<integer> :] <integer>
<dimensions> ::= * | <dimensions> , *
<type> ::= ≤<type-1> | {vector (<number>) | array
   ({<bound-pairs-1> | <dimensions>})}≥ | string
   (<number>)
<back> ::= back | <back> all
<special-modifier> ::= static | local | global | <back>
<specials> ::= <special-modifier> | <specials>
   <special-modifier>
<initializations> ::= <identifier> [:= <expression>] |
   <initializations> , <identifier> [:= <expression>]
<normal-declaration> ::= ≤<type> | <specials>≥
   <initializations>
<identifier-list> ::= <identifier> | <identifier-list> ,
   <identifier>
<user-type-declaration> ::= type <identifier>
   (<identifier-list>)
<equ-declaration> ::= equ <initializations>
<fields> ::= <type-1> <identifier-list> | <fields> ;
   <type-1> <identifier-list>
<record-declaration> ::= record <identifier>
   [ ([<fields>]) ]
<parameters> ::= ≤<type> | <specials>≥ <identifier-list>
   | <parameters> ; ≤<type> | <specials>≥
   <identifier-list>
<multiple-1) ::= <simple-type> | <compound-type> | ref
   [ (<identifier>) ] <compound-type> | <multiple-1>
   <compound-type>
<multiple> ::= {<multiple-1> s | refs [ (<identifier>) ]}
```

```
   <identifier>
<pars> ::= ([ {<parameters> [; <multiple>] | <multiple>} ])
<procedure-declaration> ::= proc <identifier> [<pars>];
   <block> corp
<typed-procedure-declaration> ::= <type-1> proc
   <identifier> [<pars>] ; <expression>
<dummy-specification> ::= [<type-1>] dummy <identifier>
   [<pars>]
<procedure-equivalence> ::= proc <identifier> =
   <identifier>
<declaration> ::= <normal-declaration> |
   <user-type-declaration> | <equ-declaration> |
   <record-declaration> | <procedure-declaration> |
   <typed-procedure-declaration> | <dummy-specification>
   | <procedure-equivalence>
```

Expressions

```
<location> ::= <any-identifier> | & <loc-expression> |
   <ref-expression> @ <int-expression> |
   <array-expression> (<int-expression-list>) |
   <string-expression> (<int-expression> [<bar>
   <int-expression>]) | <bits-expression>
   (<int-expression> [<bar> <int-expression>]) |
   {<vector-expression> | <list-expression> |
   <chain-expression> | <pair-expression> |
   <triple-expression> | <stack-expression> |
   <area-expression> | <queue-expression>}
   (<int-expression>)
   | <table-expression> (<expression>) |
   <{<stack-expression> | <queue-expression>}
<expression> ::= (<block-expression>) | <prefix>
   <expression> | <expression> <infix> <expression> |
   <assignment> | [/] / <type-1> <expression> |
   {<any-identifier> | & <procval-expression>}
   [ ([<expression-list>]) ] | <T-special-expression> |
   <T-expression>
<bar> ::= |
<T-expression> ::= <expression>
<T-block-expression> ::= <block-expression>
<expression-list> ::= <expression> | <expression-list> ,
   <expression>
<T-expression-list> ::= <T-expression> |
   <T-expression-list> , <T-expression>
<free-expression> ::= <T-expression>
<any-expression> ::= <user-type-expression>
<assignment> ::= <location> := <expression>
<fixed-loc-expression> ::= <int-loc-expression> |
   <real-loc-expression> | <bits-loc-expression> |
   <bool-loc-expression>
<ptr-loc-expression> ::= any other <T-loc-expression>
<prefix> ::= <any-identifier>
<infix> ::= <any-identifier>
<block-expression> ::= [<block-body> ;] <expression> |
     <identifier-X> <block-body> ; <expression>
     <identifier-X>
```

Statements

```
   <empty> ::=
   <declarations> ::= <declaration> | <declarations> ;
      <declaration>
   <statements> ::= <statement> | <statements> ; <statement>
   <block-body> ::= [<declarations> ;] <statements>
   <block> ::= <block-body> | <identifier-X> <block-body>
      <identifier-X>
   <statement> ::= do <block> od | <assignment> |
      {<any-identifier> | & <procval-expression>}
      [ ([<expression-list>]) ] | <special-statement> |
      <empty>
```

Integers

```
   <int-prefix> ::= - | <bar> | sign
   <int-infix> ::= + | - | * | ** | div | rem
   <int-expression> ::= <expression> | <integer> |
      <int-prefix> <int-expression> | <int-expression>
      <int-infix> <int-expression>
```

Reals

```
   <real> ::= {<integer> . [<number>] | . <number>} [E
      <integer>] | <integer> E <integer>
   <real-prefix> ::= - | <bar>
   <real-infix> ::= + | - | * | ** | /
   <real-expression> ::= <expression> | <real> |
      <real-prefix> <real-expression> | <real-expression>
      <real-infix> <real-expression> | <real-expression> **
      <integer-expression>
   <int-expression> ::= sign <real-expression>
```

Strings

```
   <quote-1> ::= '
   <quote-2> ::= "
   <char-1> ::= any character except <quote-1>
   <char-2> ::= any character except <quote-2>
   <chars-1> ::= <char-1> | <chars-1> <char-1>
   <chars-2> ::= <char-2> | <chars-2> <char-2>
   <string> ::= <quote-1> <chars-1> <quote-1> | <quote-2>
      <chars-2> <quote-2>
   <string-expression> ::= <expression> | <string> |
      <string-expression> + <string-expression>
   <int-expression> ::= size <string-expression>
```

Bits

```
   <binary-digit> ::= 0 | 1
```

```
<octal-digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
<hex-digit> ::= <digit> | A | B | C | D | E | F
<binary> ::= ¢ <binary-digit> | <binary> <binary-digit>
<octal> ::= ! <octal-digit> | <octal> <octal-digit>
<hexadecimal> ::= ! <hex-digit> | <hexadecimal>
   <hex-digit>
<bits> ::= <binary> | <octal> | <hexadecimal>
<bits-infix> ::= <bar> | & | *
<bits-expression> ::= <expression> | <bits> | ¬
   <bits-expression> | <bits-expression> <bits-infix>
   <bits-expression> | % <expression>
```

Booleans

```
<boolean> ::= true | false
<bool-infix> ::= & | * | <bar>
<comp> ::= < | <= | > | >=
<equal-comp> ::= = | ¬=
<compound-comp> ::= == | ¬==
<bool-expression> ::= <expression> | <boolean> | ¬
   <bool-expression> | <bits-expression>
   (<int-expression>)
   | <bool-expression> <bool-infix> <bool-expression>
   | <int-expression> <comp> <int-expression>
   | <real-expression> <comp> <real-expression>
   | <string-expression> <comp> <string-expression>
   | <T-expression> <equal-comp> <T-expression>
   | <T-expression> <compound-comp> <T-expression>
```

Linear Compound Objects

```
<element> ::= <sequence> | [:] <expression>
<elements> ::= <empty> | <elements> <element>
<sequence> ::= ( <elements> )
<left-bracket> ::= <
<right-bracket> ::= >
<expression-list-1> ::= [<expression>] |
   <expression-list-1> , [<expression>]
<linear> ::= <left-bracket> <expression-list-1>
   <right-bracket>
```

Pairs, Triples and Vectors

```
<pair-element> ::= <pair-1> | [:] <expression>
<pair-1> ::= ( [<pair-element> <pair-element>] )
<pair-2> ::= <left-bracket> [<expression>] ,
   [<expression>] <right-bracket>
<pair-expression> ::= P <pair-1> | P <pair-2>
<triple-element> ::= <triple-1> | [:] <expression>
<triple-1> ::= ( [<triple-element> <triple-element>
   <triple-element>] )
<triple-2> ::= <left-bracket> [<expression>] ,
   [<expression>] , [<expression>] <right-bracket>
<triple-expression> ::= T <triple-1> | T <triple-2>
```

```
<vector> ::= V <sequence> | V <linear>
<vector-expression> ::= <expression> | <vector> |
   <vector-expression> + <vector-expression>
```

Sets, Bags, Lists and Chains

```
<set> ::= S <sequence> | [S] <linear>
<bag> ::= B <sequence> | B <linear>
<list> ::= L <sequence> | L <linear>
<chain> ::= C <sequence> | C <linear>
<set-infix> ::= + | * | -
<bag-infix> ::= + | - | & | <bar>
<list-infix> ::= + | *
<chain-infix> ::= + | *
<set-expression> ::= <expression> | <set> |
   <set-expression> <set-infix> <set-expression>
<bag-expression> ::= <expression> | <bag> |
   <bag-expression> <bag-infix> <bag-expression>
<list-expression> ::= <expression> | <list> |
   <list-expression> <list-infix> <list-expression>
   | - <list-expression>
<chain-expression> ::= <expression> | <chain> |
   <chain-expression> <chain-infix> <chain-expression>
   | - <chain-expression>
<membership> ::= <- | ¬<-
<set-comp> ::= < | <= | > | >=
<bag-comp> ::= < | <= | > | >=
<bool-expression> ::=
   <expression> <membership> <set-expression> |
   <expression> <membership> <bag-expression>
   | <set-expression> <set-comp> <set-expression>
   | <bag-expression> <bag-comp> <bag-expression>
```

Stacks and Queues

```
<bool-expression> ::= empty {<stack-expression> |
   <queue-expression>}
<statement) ::= pop {<stack-expression> |
   <queue-expression>}
   | add <expression> to <queue-expression>
   | push <stack-expression>
```

References

```
<type-infix> ::= is | isnt
<bool-expression> ::= <expression> <type-infix>
   <int-expression>
```

Labels

```
<statement> ::= <identifier> : <statement> |
   {goto | go to} <label-expression>
```

Conditional Constructs

```
<elifs> ::= <empty> | <elifs> elif
   <bool-block-expression> then <block>
<if-statement> ::= if <bool-block-expression> then
   <block> <elifs> [else <block>] fi
<T-elifs> ::= <empty> | <T-elifs> elif
   <bool-block-expression> then <T-block-expression>
<T-if-expression> ::= if <bool-block-expression> then
   <T-block-expression> <T-elifs> else <T-expression>
<case-list-T> ::= <T-expression> : <block> |
   <case-list-T> , <T-expression> : <block>
<block-list> ::= <block> | <block-list> , <block>
<case-statement> ::= case <T-block-expression> of
   <case-list-T> [else <block>] esac
   | case <int-block-expression> of <block-list> [else
   <block>] esac
<T1-case-list-T2> ::= <T2-expression> :
   <T1-block-expression> | <T1-case-list-T2> ,
   <T2-expression> : <T1-block-expression>
<T-block-expression-list> ::= <T-block-expression> |
   <T-block-expression-list> , <T-block-expression>
<T-case-expression> ::= case <T2-block-expression> of
   (<T-case-list-T2> [else <T-block-expression>])
   | case <int-block-expression> of
   (T-block-expression-list> [else <T-block-expression>])
<cond-list> ::= <bool-block-expression> : <block> |
   <cond-list> , <bool-block-expression> : <block>
<cond-statement> ::= cond <cond-list> [else <block>] dnoc
<T-cond-list> ::= <bool-block-expression> :
   <T-block-expression> | <T-cond-list> ,
   <bool-block-expression> : <T-block-expression>
<T-cond-expression> ::= cond (<T-cond-list> [else
   <T-block-expression>])
<special-statement> ::= <if-statement> | <case-statement>
   | <cond-statement>
<T-special-expression> ::= <T-if-expression> |
   <T-case-expression> | <T-cond-expression>
```

Iterative Constructs

```
<for-part> ::= for <location> {
   [from <int-block-expression>] [to
   <int-block-expression>]
   [by <int-block-expression>] |
   [from <real-block-expression>]
   [to <real-block-expression>] [by
   <real-block-expression>]
   | [@] in <compound-block-expression>[1]
   | via <expression> }
   [suchthat <bool-block-expression>]
```

---

[1] where "compound" is to be replaced by set, bag, list, chain, vector, array, table, stack or queue

```
<for-parts> ::= <empty> | <for-parts> <for-part>
<iteration-statement> ::= <for-parts> [while
   <bool-block-expression>] do <block> [until
   <bool-block-expression>] od
<T-iteration-expression> ::= <for-parts> [while
   <bool-block-expression>] using
   {<T-procval-block-expression> | <T-proc-identifier>}
   collect <expression> [until <bool-expression>]
<special-statement> ::= <iteration-statement> | return
   [<expression>] | exit [<identifier>] | iterate
   [<identifier>] | next [<identifier>]
<T-special-expression> ::= <T-iteration-expression>
```

Input and Output

```
<location-list> ::= <location> | <location-list> ,
   <location>
<special-statement> ::= card ( <location-list> ) |
   line ( <expression-list> ) | inbuff ( <location> )
   | outbuff ( <expression> )
```

The Data Net

```
<char3> ::= any character except blank
<chars3> ::= <char3> | <chars3> <char3>
<dict-expression> ::= . <chars3>
<element-1> ::= <sequence-1> | <chars3> | [:]
   <expression>
<elements-1> ::= <empty> | <elements-1> <element-1>
<sequence-1> ::= ( <elements-1> )
<D-expression> ::= D <sequence-1> | D <linear>
```

Nonstandard Evaluation

```
<locator> ::= <identifier> | <locator> # <number> |
   <identifier> # <locator>
<expr-expression> ::= # <expression> | <locator>
<empty-statement> ::= ¢ <statement> | + <statement>
<expression> ::= = <expr-expression>
```

Patterns and Pattern Matching

```
<pattern-prefix> ::= ¬ | <- | <-> | ->
<pattern-infix> ::= & | * | <bar>
<pat-bool-prefix> ::= :<- | :<-> | :->
<pattern> ::= <expression> | <pattern-prefix> <pattern>
   | <pattern> <pattern-infix> <pattern>
   | <pat-bool-prefix> <bool-expression> |
   <pattern> {@ | ¢} [<int-expression>] [C=<location>]
   | <pattern> = [@] <location>
   | - | -- | ---
<sequence-prefix> ::= <simple-type> | <compound-type> | S
   | B | V | L | C | P | T
<element-2> ::= [<sequence-prefix>] <sequence-2> |
   <chars3> | [:] <expression> | <pattern>
<elements-2> ::= <empty> | <elements-2> <element-2>
<sequence-2> ::= ( <elements-2> )
<pat-expression> ::= D <sequence-2> | <D-expression>
<with> ::= with <pat-expression>
<withs> ::= <empty> | <withs> <with>
<T-special-expression> ::= {find | find1}
   <pat-expression> [undo] [need] <withs> [suchthat
   <bool-expression>] [in <world-expression>]
<special-statement> ::= reset <pat-expression> <withs>
   [in <world-expression>]
```

Contexts

```
<location> ::= <location> wrt <context-expression>
```

Processes

```
<resume> ::= resume <process-expression> [with
   <expression>] [time <int-expression>]
<process-expression> ::= incarnate
   <procval-block-expression> as <int-expression>
   [priority <int-expression>]
<T-special-expression> ::= <resume>
<special-statement> ::= <resume>
<event-special-expression> ::= wait
   <event_type-expression> [priority <int-expression>]
```

Events and Event_types

```
<process-list> ::= <process-expression> | <process-list>
   , <process-expression>
<event_type-expression> ::= {active | inactive}
   <process-expression> | {any | all} {active | inactive}
   {(<process-list>) | <process list-expression>}
<special-statement> ::= cause <event_type-expression>
   [with <expression>] | on <event_type-block-expression>
   do <block> od
```

Backtracking

```
<try> ::= <for-parts> [while <bool-block-expression>] try
   [<string>]
<ptry> ::= <for-parts> [while <bool-block-expression>]
   ptry <block> then [<string>]
<process list loc-special-expression> ::= <ptry>
<attempt-list> ::= <block> : <block> | <attempt-list> ,
   <block> : <block>
<attempt> ::= attempt <attempt-list> [else <block>] end
<fresume> ::= fresume <process-expression> [time
   <int-expression>]
<T-special-expression> ::= <fresume>
<special-statement> ::= <try> | <attempt> | <fresume>
   | failing [<string>] <statement>
```

Worlds

```
   <special-statement> ::= enter <world-expression> [wrt
      <context-expression>] | delete {<expression> |
      <D-expression> [in <world-expression>]} [wrt
      <context-expression>]
   <instance-special-expression> ::= assert <D-expression>
      [in <world-expression>] [wrt <context-expression>]
```

Elaborations

```
   <special-statement> ::= <identifier> | <expression> [as
      <expr list-expression>] | <elab-expression>
```

Appendix B - Internal Structures

B1 Language Defined Record Classes

Of the many data types in ALAI, some use only a single value, while others, like contexts, tables, lists, etc. use several within the single entity. The presence of the general type 'free' (needed to make the data net general but still accessible through the standard language constructs) requires that all types be represented by some single, uniform convention. The simplest and most common solution is to represent multi-valued entities by a single pointer to a record containing the various values. Proper use of type 'free' requires that the type of such records be determinable, hence some type code must accompany all values to be assigned to 'free' locations. (For uniformity, the same convention is extended to locations of types 'any' and 'ref'.) This type code can be either a part of the pointer or a part of the record pointed at, but for ALAI, the choice has been made: it goes with the pointers. For single value types, the value is stored directly, rather than being pointed to. The fields of the various language defined record classes (i.e. the data types with more than one value in them) are as follows:

- bool - type code 1
- bits - type code 2
- int - type code 3
- real - type code 4
- loc - type code 5
- pair - type code 6
  - free *LEFT
  - free *RIGHT
- triple - type code 7
  - free *LEFT
  - free *MIDDLE
  - free *RIGHT
- list - type code 8
  - free *VALUE
  - list *NEXT
- set - type code 9
  - free *VALUE
  - set *NEXT
- bag - type code 10
  - free *VALUE
  - bag *NEXT
- stack - type code 11
  - free *VALUE
  - stack *NEXT
  - A stack pointer points to the top of the stack. The links in the stack point away from the top. The bottom node has a null *NEXT.
- queue - type code 12
  - free *VALUE

queue *NEXT

A queue pointer is to a special record of type 12. Its *VALUE points to the node at the back of the queue, its *NEXT to the node at the front. The links go from front to back. *NEXT of the back node is null.

chain - type code 13
- chain *PREVIOUS
- free *VALUE
- chain *NEXT

area - type code 14
- int *LEN
- ? *REGION

*LEN is the number of storage units in the region pointed to by the untyped pointer *REGION. The count will probably be in some handy unit such as byte or word.

string - type code 15
- int *LEN
- area *REGION

*LEN is the number of characters currently in the string. *REGION is the area (which contains its own length) of storage in which the characters are located.

vector - type code 17
- int *LEN
- area *REGION

*LEN is the number of elements in this vector. *REGION is the area (whose length may be greater than is necessary) in which the elements are stored.

array - type code 19
- int pair list *BOUNDS
- area *REGION

The pairs in *BOUNDS are the upper and lower bounds for the corresponding dimensions of the array.

table - type code 22
- int *LEN
- area *REGION

*LEN is the current number of entries in the table. The entries are two consecutive values arrayed through *REGION: the index value, followed by the corresponding table entry.

label - type code 23
- int *BLOCKLEV
- free *WHERE

*BLOCKLEV is the block level of the block containing the definition of the label. It is used to control block exit when branching. *WHERE will be either an 'expr' pointer to what statement to interpret next or a 'code' pointer to the instruction to branch to.

elab - type code 25
- free *OBJECT
- list *METHODS
- int *ELABNO
- I list *KNOW

context - type code 27
- context *PARENT

context list *CHILDREN
world *BIRTHPLACE
int *LEVEL
free *CHANGES
free *ROOTER

Contexts were discussed previously except for the structure of *CHANGES and *ROOTER. *CHANGES is the head of a circular linked list of "change records" (type code 56) which have five fields, all of which are nominally of type 'free'. Their usage is quite restricted however. The first field is the link to the next such record or back to the context (type code 56 or 27). The second is essentially a 'loc' pointer to the location whose value is being saved by this record, but its type field (for the type of the thing it contains or points at) has been converted to a boolean flag (it is not a proper boolean, it is only used that way) indicating whether or not this record is in the *CHANGES list of the current context. (To reference it, use e.g. (%type(X))(*L).) The third field is a part of a ring of fields which leads ultimately to the fourth field of the "change record" which holds the previous value for the location being saved. In this ring will be type 58 pointers to the third field of "change records", exactly one type 57 pointer to the fourth field of a "change record" and possibly type 59 or 60 pointers to the second or third field of "rooter records". These rings, using only one pointer per record, provide two-way links between a saved value and all subsequent values which are dependent on it. The fifth field is the properly typed value being saved by this record. *ROOTER is the head of a circularly linked list of "rooter records" (type code 61) which have three 'free' fields. The first is a type 61 pointer to the next record or a type 27 pointer back to the context. The second and third fields participate in the above-mentioned rings: the second field ultimately points to the value for the location under *ROOT, the third field to its value under the context in question. Such *ROOTER lists are usually longer than *CHANGES lists.

process - type code 28
area stack list *BASES
free *CONTINUE (actually 'code' or 'expr')
area *SAVEAREA
int *STATUS
free *PATH (types 64-67)
event_type *WAITFOR
event_type list *TRIGGERA
event_type list *TRIGGERI
int *PRIORITY
context *PROCONTEXT
world *PROWORLD
elab stack *OBJECTS
pair list *FIXLIST
list *FINDLIST

event_type - type code 29
    process queue *WAIT
    event queue *NOTICE
    free *WHAT
    I list *KNOW
event - type code 30
    free *MESSAGE
    event_type *TYPE
    I list *KNOW
C - type code 31
I - type code 32
N - type code 33
D - type code 34
CD - type code 35
ID - type code 36
ND - type code 37
dict - type code 41
    string *NAME
    list *MEANINGS
pat - type code 42
    a sub-type of 'desc'
expr - type code 43
    a sub-type of 'desc'
code - type code 44
    used to point to sections of machine code
world - type code 45
    dict table *DICTIONARY
    context *CURCONT
    context *WROOT

*DICTIONARY is a table indexed by the strings which are the names of the objects in this world. Its entries are the corresponding 'dict' records. *CURCONT is the "current context" for use with this world. *WROOT is the root of the context tree for this world.

procval - type code 46
    int list *TYPE
    free *NAMES (actually 'dict' or 'dict set')
    int *STORSIZE
    pair table *PARS
    expr *EBODY
    code *CBODY

*TYPE is the type of the procedure, e.g. 'int list loc proc A' would yield a *TYPE of 'L<5,8,3>'. The leftmost bit of the first element of the list (the major type) controls whether to use *EBODY (0) or *CBODY (1) for this procedure. The bit second from the left in the same word (accessed via '(%((PROC@*TYPE)(1)))(2)') controls whether *NAMES is a single 'dict' pointer (0) or a set of them (1). *EBODY and *CBODY are pointers to the program structure version and the compiled version of the body of the procedure. Either may be null (as may *NAMES). *STORSIZE is the amount of storage needed for the parameters of the procedure. *PARS is a table, indexed by the string names, of pairs indicating the types and positions of the parameters of the procedure. The nature of the pairs will be discussed later (see 'Symbol Tables'). If this procedure has been compiled, then in the interests of saving storage space, both *EBODY and *PARS may be null (i.e. the program structure and the symbol table have been destroyed).

ref - type code 48
includes all user declared record classes

any - type code 51
includes all user defined data types

free - type code 54
includes all ALAI types

equ - type code 55
parse time variables

Types 'ref' - 'equ' will never occur in an actual run-time pointer (unless the programmer puts one there). Types 64 - 67 are used for block entry, method entry, incarnation and backtrack points in the backtrack tree.

B2 Internal Program Structures

Programs in ALAI can be either compiled or uncompiled. Compiled programs are represented by a single pointer to the top of the machine code. Uncompiled programs, which are executed via an interpreter, are represented as networks of descriptors which use special language defined concepts as the relation (*REL). Note that all participants of 'expr' nodes are actually 'free'. The concepts used and the participants expected are as follows:

*BLOCK used to represent all block expressions and statements
- *PAR1: the name (if any) given to the block
- *PAR2: int - the storage needed for this block's locals
- *PAR3: pair table - its symbol table (see later)
- *PAR4: list - list of the statements (and expression) of the block

*ASSIGN used for all assignments
- *PAR1: the location being assigned to
- *PAR2: the expression being assigned

*REF used for all identifier references
- *PAR1: the symbol table entry
- *PAR2: the list of subexpressions
These can be indices, arguments or a field selector.

*IF used for if statements and expressions
- *PAR1: the original 'if' boolean expression
- *PAR2: the original true case statements or expression
- *PAR3: a list of 'elif's. Each is a pair: a boolean expression and a list of statements (and expression).
- *PAR4: the final 'else' part

In the *IF form and in others, where one statement or expression is accepted, a list can appear, the elements of which are to be serially executed. The list represents a block statement or expression without declarations.

*CASE1 used for indexable case constructs
   *PAR1: the indexing expression
   *PAR2: a list of the expressions or statements
   *PAR3: the 'else' part
*CASE2 used for key-compare case constructs
   *PAR1: the key expression
   *PAR2: a list of pairs: the compare expression and the resulting statement or expression
   *PAR3: the 'else' part
*COND used for 'cond' constructs
   *PAR1: a list of pairs: boolean expression and the resulting statement or expression
   *PAR2: the 'else' part
*STEP used for range specification in 'for' constructs
   *PAR1: 'from' expression
   *PAR2: 'to' expression (null for infinity)
   *PAR3: 'by' expression
*IN used for 'in' specifications in 'for's
   *PAR1: the 'in' expression
*ATIN used for '@in' specifications
   *PAR1: the '@in' part
*VIA used for 'via' specifications in 'for's
   *PAR1: the 'via' expression
*FOR-PART used for <for-part>'s in 'for's
   *PAR1: the 'for' variable
   *PAR2: *STEP, *IN, *ATIN or *VIA
   *PAR3: the 'suchthat' expression
*FORS used for iteration statements
   *PAR1: a list of *FOR-PART's
   *PAR2: the 'while' expression
   *PAR3: the block or statement list (the body)
   *PAR4: the 'until' expression
*FORE used for iteration expressions
   *PAR1: a list of *FOR-PART's
   *PAR2: the 'while' expression
   *PAR3: the 'using' expression
   *PAR4: the 'collect' expression
   *PAR5: the 'until' expression
*RETURN used for 'return' statements
   *PAR1: the value to be returned (if any)
*EXIT used for 'exit' statements
   *PAR1: the 'for' to exit (its string label)
*NEXT used for 'next' statements
   *PAR1: the label of the 'for' to iterate
*ITERATE used for 'iterate' statements
   *PAR1: the label of the 'for' to iterate within
*CARD used for 'card' statements
   *PAR1: list of the "argument" locations and statements
*LINE used for 'line' statements
   *PAR1: list of the "argument" expressions and statements
*INBUFF used for 'inbuff' statements
   *PAR1: the location to read to
*OUTBUFF used for 'outbuff' statements
   *PAR1: the expression to output

*TF used for the '<' stack and queue operator
   *PAR1: the stack or queue to take the top or front element of
*DEREF used for dereferencing 'loc's
   *PAR1: the 'loc' expression to be used as a location
*EXEC used for calling 'procval's
   *PAR1: the 'procval' expression to be called
   *PAR2: the list of argument expressions
*GOTO used for branching ('goto')
   *PAR1: the 'label' expression to branch to
*TYPE used to request run-time type checking
   *PAR1: the integer list specifying the required type
   *PAR2: the expression whose type is to be checked
*UNEVAL used to produce 'expr's
   *PAR1: the expression to be used as an 'expr'
*EVAL used to represent the '=' prefix operator
   *PAR1: the 'expr' expression to be evaluated
*NOT, *LEFT, *BOTH, *RIGHT represent pattern prefixes ¬, <-, <-> and ->
   *PAR1: the pattern the prefix is to apply to
*AND, *OR and *XOR represent the pattern infixes &, | and *
   *PAR1: the list of subpatterns being combined
*BLEFT, *BBOTH and *BRIGHT represent the prefixes :<-, :<-> and :->
   *PAR1: the boolean expression to be tested
*SPAN represents the '@' pattern postfix
   *PAR1: the integer count needed (null for not given)
   *PAR2: the location to put the count into
*ARBNO represents the '¢' pattern postfix
   *PAR1: the integer count needed (null for not given)
   *PAR2: the location to put the count into
*PASSIGN represents the '=' pattern operator
   *PAR1: the pattern to match
   *PAR2: the location to assign the target to
*PPOS represents the '=@' pattern operator
   *PAR1: the pattern to match
   *PAR2: the location to assign the position number to
*P, *T, *V, *L, *C, *S, *B, *ST and *BIT are used to represent pairs, triples, vectors, lists, chains, sets, bags, strings and bit sequences as pattern structures
   *PAR1: the sequence which this is an element of
   *PAR2: the chain of elements for this sequence
*TYPEARB used to represent restricted 'arb' values
   *PAR1: the integer type of the required part of the target
*INT used to represent integer comparison patterns
   *PAR1: the integer comparison code for the test (see 'comp' under 'Booleans')
   *PAR2: the integer expression used as right-hand comparand
*REAL used to represent real comparison patterns
   *PAR1: the integer comparison code
   *PAR2: the real expression used as right-hand comparand
*FIND used for the 'find' expression

*PAR1: the pattern to be found
*PAR2: a list of the 'with' patterns
*PAR3: the 'suchthat' boolean expression
*PAR4: the 'in' world expression
*PAR5: bit 1: 1 for 'need'
*FIND1 used for the 'find1' expression
as *FIND
*RESET used for the 'reset' statement
*PAR1: the pattern searched for
*PAR2: a list of the 'with's used
*PAR3: the 'in' world expression
*WRT used for the 'wrt' construct
*PAR1: the location being referenced
*PAR2: the context to reference with respect to
*RESUME used for the 'resume' construct
*PAR1: the process to resume
*PAR2: the 'with' expression to pass
*PAR3: the 'time' int-expression to use
*FRESUME used for the 'fresume' construct
*PAR1: the process to resume
*PAR2: the 'time' int-expression to use
*INCARNATE used for the incarnate expression
*PAR1: the procval expression to use as body
*PAR2: the integer 'as' expression
*PAR3: the integer 'priority' expression
*WAIT used for the 'wait' expression
*PAR1: the event_type to wait for
*PAR2: the integer 'priority' expression
*ACTIVE, *INACTIVE, *ALLACTIVE, *ALLINACTIVE, *ANYACTIVE, *ANYINACTIVE see the discussion under 'Events and Event_types'
*CAUSE used for the 'cause' statement
*PAR1: the event_type to be caused
*PAR2: the 'with' message to be passed
*ON used for the 'on' statement
*PAR1: the event_type to be trapped
*PAR2: the block or list of statements to do on the trap
*TRY used for the 'try' statement
*PAR1: a list of *FOR-PART's
*PAR2: the 'while' boolean expression
*PAR3: the string label for this 'try'
*PTRY used for the 'ptry' expression
*PAR1: a list of *FOR-PART's
*PAR2: the 'while' boolean expression
*PAR3: the 'ptry' body block
*PAR4: the string label for this 'ptry'
*ATTEMPT used for the attempt statement
*PAR1: list of pairs: block to try, block to do on success
*PAR2: the 'else' part
*FAILING used to represent the 'failing' statement
*PAR1: the statement to execute when failed back to
*PAR2: the string label for this 'failing'
*ENTER used for the 'enter' statement

*PAR1: the world to enter
*PAR2: the context to enter with respect to
*DELETE used for the 'delete' statement
*PAR1: the expression or pair (D, world) to delete
*PAR2: the context to delete with respect to
*ASSERT used for the 'assert' expression
*PAR1: the descriptor of what to assert
*PAR2: the world to assert in
*PAR3: the context to assert with respect to

B3 Symbol Tables

There are three types of symbol tables in ALAI, but all are structured the same. The first type is the global symbol table; it contains declarations for all language defined names and all fixed names (declared by the user but not within any procedures). The second type are those which are part of blocks (*BLOCK@*PAR3); they are essentially the same as the global symbol table, but are associated with particular blocks. The third type are those which are part of a procedure (*PARS); they are similar to the other types but will not contain constants, record class descriptions or user data type descriptions. The fixed variable *GLSYMTAB points to the global symbol table. All symbol tables are indexed by the strings which are the names of the entries and contain free pairs. The first element of the pairs is an integer or a list giving the type of the identifier. The most major type is the first in the list. Bits 0 and 1 of the major type indicate whether the identifier is static (0) or dynamic (1) and local (0) or global (1) respectively. In each level of the type which is used as an integer type, bits 2 and 3 indicate whether or not the identifier is safe (1) and back (1) respectively at that level. For standard variables, the second element of the pair indicates the displacement of that variable in either the static or the dynamic storage region. Special cases are as follows:

fixed size string - type code 16
The last element in the type list will be the length in characters of the string (the amount of storage actually reserved is irrelevant).

fixed size vector - type code 18
The last element in the type list will be the number of elements in the vector.

fixed dimensionality array - type code 20
The element in the type list immediately following the 20 is the number of dimensions in the array.

fixed size array - type code 21
The element in the type list immediately following the 21 is a list of pairs indicating the respective lower and upper bounds of the indices of the array.

label constant - type code 24
The second element of the pair is the label constant (type 'label') which is named by this entry.

elab constant - type code 26

The second element of the pair is the elab constant (type 'elab') which is named by this entry.

proc constant - type code 47

The second element of the pair is the procedure constant (type 'procval').which is named by this entry.

ref - type code 48

The entry in the type list after the 48 is the integer type number for the record class which this variable is allowed to reference. If this integer is zero then this variable can reference any user defined record class.

record field name - type code 49

The second element of the pair is this field's offset in the record. The two elements in the type list after the 49 are a pointer to the entry for this record class and the type of this field.

record class name - type code 50

The second element of the pair is a vector of pointers to the symbol table entries for the fields of this record class. The next element of the type list is the integer type code for this record class.

any - type code 51

The entry after the 51 in the type list is the integer type number for the user defined data type which this variable can reference. If this integer is zero then this variable can reference any user defined data type.

user type constant - type code 52

The second element of the pair is the integer equivalent of this identifier. The element in the type list after the 52 is a pointer to the symbol table entry for this user defined data type.

user type class name - type code 53

The second element of the pair is a vector of pointers to the symbol table entries for the constants of this type. The element after the 53 in the type list is the integer type code for this user defined data type.

equ - type code 55

The second element of the pair is the current value of this equ.

constant - type code 62

This type is used to reserve storage for constants which are created at run time and will not change throughout the duration of the block or procedure which this entry is in, e.g. 'loc' constants. The second element of the pair holds the displacement of the pseudo-variable. The elements of the type list beyond the 62 hold the type of the constant. Such pairs are listed under the name *SPECIAL.

multiple - type code 63

This signifies that the string which indexed this pair has more than one meaning in this block. The second element of the pair is a list of the pairs which represent the various meanings in the same way that normal symbol table pairs operate.

In the global symbol table, the name *CONSTANT yields a pair whose second element is a list of fully typed constants. The presence of such constants in the symbol table prevents duplication of constants. The parser will prevent attempts to alter parts of constants, so that multiple use and re-use of them is safe.

Appendix C - Operating Environment

As has been mentioned previously, ALAI is intended to be used interactively. The overall effect is that of an ALGOL type APL. Some kind of external file would be used for storage of a "workspace" which would contain the data net, compiled and uncompiled programs, fixed variables and their values, etc. in the same state as when this "workspace file" was last updated. Information can thus be saved between runs, allowing the gradual accumulation of banks of data and sets of procedures. When ALAI is first run with a new workspace, no user procedures exist and no fixed identifiers have been declared, so what can the user do? In this mode of operation, called "base mode", declarations can be made by entering a <declaration> preceded by the word 'declare'. The objects thus defined (they can be simple variables, record classes, procedures, etc.) are entered into the global symbol table as fixed objects. They can then be referenced from within procedures and, more interestingly, from base mode. When in base mode a statement entered is executed immediately after the entire statement has been parsed. Expressions entered in base mode are treated similarly (the procedure 'ignore' is not needed, expressions themselves are accepted) and the resulting value is printed out.

The ways in which ALAI communicates with the host operating system and with the devices it supports will of course be dependent on that system and on the choices of whoever designs the interface. Possible features are those for workspace control, i.e. a library of them, switching among them, copying parts of one into another, etc. Another local feature is that of the existence of and the nature of a compiler to produce machine code for the local computer. Such a feature is not necessary, but is highly desirable in that it would greatly reduce the execution time of procedures which had been compiled. Many of the pattern features available, e.g. *SPAN, *ARBNO, etc. can be compiled into executable code rather than kept as structures which must be interpreted by a pattern matcher.

The vast majority of programmers do not have the knack of writing perfect programs on the first try, hence some way is needed whereby procedures existing in an ALAI workspace can be changed. One method is to destroy the old version and then re-enter the new one. This method is not very handy in an interactive environment, however, as it is dependent on the text editing and file handling facilities of the local system. An alternative is to edit the actual program structure within the workspace. A semantic editor (written entirely in ALAI of course) could be a part of the ALAI system. It would provide simple to use interactive facilities for searching through programs, for changing small parts, for deleting parts, for moving parts, etc. In effect it would be similar to text editors except that it would operate on internal program structures as pointed to or indicated by locators. It would be dependent on the existence and useability of the parser (also written in ALAI) and of a routine for printing out program structures in a neat, legible format (like the LISP "pretty-print" routines).

Further capabilities which are possible include the convenient use of the system as a batch processor. Fixed variables would still be at the top level of execution, but statements entered in base mode would be compiled as the main program rather than being executed and then discarded. The resulting object module should be directly executable under the host operating system. Also desirable is a means for accessing routines written in some other language (usually desired is the local assembly language or FORTRAN) so that special purposes and/or high efficiency can be achieved. This would be of particular interest to those wishing to do numerical operations, as the definition of ALAI does not include such standard functions of analysis as SIN, COS, LOG, EXP, etc. Features such as these have been standardized in other programming languages, but it is felt that the details of such should be decided by those who implement the language in a given environment, rather than by the language designer.

Appendix D - Sample Programs

An ideal sample program to demonstrate the advantages of ALAI would be one that speaks and understands English fluently and that possesses powerful and general problem solving and abstraction capabilities. Although such uses are the intent of the language, they are some time in the future, so somewhat less extravagant examples will be given. A seemingly common sample program is one with a set of procedures which "plans" and executes a solution to the monkey and bananas problem. Such a program, though easily written in ALAI, does not demonstrate much of the language. Instead the problem of the missionaries and cannibals will be used. Three missionaries and three cannibals must cross a river in a single boat (capacity two people) in such a way that the cannibals never outnumber the missionaries. One of the features of ALAI that can be seen here is the way in which programs can be similar in structure to plans that people may develop. A simple-minded plan for the M and C problem might go as follows: "Put some (1 or 2) people into the boat. Cross to the other side. Keep doing this until everybody is on the far side. Make sure that the cannibals never outnumber any missionaries present." The "keep doing until" in ALAI would be a looping statement; putting people into the boat could be transferring them from a bag representing the bank into one representing the boat; crossing the river could be switching bags; the outnumber check could be a conditional failure if the selection of passengers is done by a backtracking construct ('try'); etc. The following ALAI procedure follows the "people-plan" and can be derived from it in a very few minutes of thought (by one familiar with the language):

```
proc MISS_CAN1;
   string list ACTIONS:=L<>;
   string BANK:="left";
   int I; equ M:=simple,C:=simple;
   simple bag LEFT:=B<M,M,M,C,C,C>,
      RIGHT:=B<>,BOAT;
   simple bag loc SIDE:=@LEFT;
   bool proc BAD(simple bag X);
      num(C,X)>num(M,X)&num(M,X)¬=0;
   *ALLBACK:=*BTCONT:=true;
   do ignore ACTIONS*L<"Empty the boat.">;
      BOAT:=B<>;
      for I in <1,2> try;
      for to I do
         for X in &SIDE try;
         &SIDE:=&SIDE-B<X>;
         BOAT:=BOAT+B<X>;
         ignore ACTIONS*L<"Put a "+(if X=C
            then "cannibal" else "missionary")+
            " into the boat."> od;
      if BAD(&SIDE) | BAD(BOAT) then fail fi;
      if SIDE=@LEFT
         then SIDE:=@RIGHT; BANK:="right"
         else SIDE:=@LEFT; BANK:="left" fi;
      &SIDE:=&SIDE+BOAT;
      if BAD(&SIDE) then fail fi;
      ignore ACTIONS*L<"Cross to the "+BANK+" bank.">
      until size(RIGHT)=6 od;
   for BANK in ACTIONS do line(BANK) od
corp;
```

After a bit of study and perhaps a bit of referring back to see what some of the symbols do, this procedure should be fairly clear. ACTIONS is used to hold a sequence of sentences which tell how to solve the problem. SIDE is used to indicate either LEFT or RIGHT so that the body of the procedure need not be concerned with which particular bank it is on. *ALLBACK and *BTCONT are turned on (all locations are backtracked for all 'try' statements) for simplicity (but not for efficiency). The procedure as it stands has several problems, one of which is that it will not work; the reason being that the selection of one person (tried first) from the system ordered bags is deterministic - that one person will be shuttled back and forth endlessly. An attempt to fix this can be made by trying a load of two people first when crossing to the far (right) bank (a simple heuristic). To guarantee a solution however, it is necessary to prevent loops of any kind, i.e. no situation may ever be repeated. This requires a record of all situations which have occurred, a record nicely provided by the contexts between the current one and the one in use when the procedure was called. The procedure would still be far from perfect however; it would waste much time trying equivalent combinations, it would waste time trying combinations which are doomed to fail, it is inflexible in what it can solve, etc. It is also very sloppy - it disregards the needs of its caller by changing vital flags and it creates and switches down a large tree of contexts. It also sets up and leaves many backtrack points which can trap a fail having nothing to do with them. If all of these loose ends are tied up it may also be wise to generalize on its capabilites by making parameters out of the numbers of people, the numbers who can row, the capacity of the boat, how many it takes to row the boat, how many solutions are required, etc. If these features are taken into account, the following procedure can be obtained (it has not been tested as no ALAI implementation yet exists):

```
proc MISS_CAN2(int CODE,BOATLOAD,CREW,RM,RC,NRM,NRC);
   int vector LEFT:=V<RM,RC,NRM,NRC>, RIGHT:=V<0,0,0,0>;
   int vector loc back backallall SIDE:=@LEFT;
   int back LOAD,B1,B2,B3,B4,B1A:=0,B2A:=0,B3A:=0,B4A:=0;
   int list COMEBACK:=for I from CREW to BOATLOAD using
      listconc collect L<I>, GOOVER:=listreverse COMEBACK;
   int list loc back MOVE:=@GOOVER;
```

```
    string back BANK;
    string list list ACTIONS:=L<>;
    string list backall ACTS:=L<>;
    string list BEST;
    context M,N,C:=*MYCONTEXT;
    context list CL:=*MYCONTEXT@*CHILDREN;
    bool vector(2) SAVE:=V<*BTCONT,*ALLBACK>;
    int proc MIN(ints X);
       (int T:=<X;
       while ¬empty X do
          if T><X then T:=<X fi;
          pop X od;
       T);
    int proc MAX(ints X);
       (int T:=<X;
       while ¬empty X do
          if T<<X then T:=<X fi;
          pop X od;
       T);
    proc CDESTROY(context X);
       context C;
       for C in X@*CHILDREN do CDESTROY C od;
       destroy X
    corp;
    if CODE<1|CODE>3|BOATLOAT<0|CREW<0|RM<0|RC<0
          |NRM<0|NRC<0 then
       line("Invalid problem specification - solution ",
          "not attempted.");
       return fi;
    if BOATLOAD<2|CREW>RM+RC|RM+NRM<RC+NRC then
       line("No solution to this problem.");
       return fi;
    *ALLBACK:=false; *BTCONT:=true;
    failing "point" do
       if size(ACTIONS)=0 then
          line("No solution to this problem.")
       elif CODE=2 then
C Print out all possible solutions
          for ACTS in ACTIONS for LOAD by 1 do
             line("Solution number "+repint(LOAD,-1)+":");
             for BANK in ACTS do line(BANK) od od
C Print solution with minimum number of boat trips
       else LOAD:=size BEST:=ACTIONS(1);
          for ACTS in ACTIONS do
             if (B1:=size ACTS)<LOAD then
                LOAD:=B1;
                BEST:=ACTS fi od;
          line("One best solution is:");
          for BANK in BEST do line(BANK) od
       fi;
       goto EXIT od;
    while LEFT(1)>0|LEFT(2)>0|LEFT(3)>0|LEFT(4)>0 do
       ignore ACTS*L<"Empty the boat.">;
       for LOAD in &MOVE suchthat
```

```
C Don't try doomed boatloads
         LOAD<=(&SIDE)(1)+(&SIDE)(2)+(&SIDE)(3)+(&SIDE)(4)
      try;
      for B1 from MAX(CREW-(&SIDE)(2),LOAD-(&SIDE)(2)-
            (&SIDE)(3)-(&SIDE)(4),0)
C Must be enough others to row and fill the boat
C Don't overload or take more than there are
         to MIN(LOAD,(&SIDE)(1)) try;
      for B3 from MAX(LOAD-B1-(&SIDE)(2)-(&SIDE)(4),0)
         to MIN(LOAD-B1,(&SIDE)(3))
      suchthat (B4:=B1+B3=0|2*B4>=LOAD)&
         ((&SIDE)(1)=B1&(&SIDE)(3)=B3|(&SIDE)(1)+
         (&SIDE)(3)-2*B4>=(&SIDE)(2)+(&SIDE)(4)-LOAD) try;
C Can't be negative and must leave enough for LOAD
C Don't take too many or more than there are
C Don't let missionaries be outnumbered in boat or on bank
      for B2 from MAX(CREW-B1,LOAD-B1-B3-(&SIDE)(4),0)
         to MIN(LOAD-B1-B3,(&SIDE)(2)) try;
C Enough to man and fill the boat
C But not overload it or take more than there are
      B4:=LOAD-B1-B2-B3;
      if B1=B1A&B2=B2A&B3=B3A&B4=B4A then fail fi;
C Don't undo what was just done
      (&SIDE)(1):=(&SIDE)(1)-B1; (&SIDE)(2):=(&SIDE)(2)-B2;
      (&SIDE)(3):=(&SIDE)(3)-B3; (&SIDE)(4):=(&SIDE)(4)-B4;
      if SIDE=@LEFT then
         SIDE:=@RIGHT;
         MOVE:=@COMEBACK;
         BANK:="right" else
         SIDE:=@LEFT;
         MOVE:=@GOOVER;
         BANK:="left" fi;
      if LOAD:=(&SIDE)(1)+(&SIDE)(3)+B1+B3<
         (&SIDE)(2)+(&SIDE)(4)+B2+B4&LOAD¬=0 then fail fi;
C Fail if would be disaster on arrival at opposite bank
      (&SIDE)(1):=(&SIDE)(1)+B1; (&SIDE)(2):=(&SIDE)(2)+B2;
      (&SIDE)(3):=(&SIDE)(3)+B3; (&SIDE)(4):=(&SIDE)(4)+B4;
C Now check that this situation is a new one
      M:=*MYCONTEXT;
      if for M via (
            for LOAD to 8 do M:=M@*PARENT
            until M=C od;   M)
         suchthat M¬=C
         using bor collect
            (for LOAD to 4 using band collect
               LEFT(LOAD)=LEFT(LOAD) wrt M)
         until M=C then fail fi;
      B1A:=B1; B2A:=B2; B3A:=B3; B4A:=B4;
      ignore ACTS*
         L<"Put "+repint(B1,-1)+"rowing missionaries, "+
         repint(B2,-1)+" rowing cannibals, "+repint(B3,-1)+
         " non-rowing missionaries and "+repint(B4,-1)+
         " non-rowing cannibals into the boat.",
         "Cross to the "+BANK+" bank.">
```

```
        od;
C Save the solution we have found
    if CODE¬=1 then ignore ACTIONS*L<copy ACTS>; fail fi;
    line("One solution is:");
    for BANK in ACTS do line(BANK) od;
C Remove created backtrack points
EXIT:succeed "point";
    switch C;
    *BTCONT:=SAVE(1); *ALLBACK:=SAVE(2);
C Restore flags and destroy spurious contexts
    for M in *MYCONTEXT@*CHILDREN do
        if for N in CL using band collect M¬=N
            then CDESTROY M fi od;
C Parse time: permit tampering
    +*PERMIT:=true;
    destroy *MYCONTEXT@*CHILDREN;
    *MYCONTEXT@*CHILDREN:=CL;
    +*PERMIT:=false
corp;
C   To find one solution to the standard problem:
MISS_CAN(1,2,1,3,3,0,0);
```

MISS_CAN2 is fairly lengthy and, because of the power of the ALAI constructs used, it is quite complex. The seven parameters are as follows:

CODE - if 1 then print first solution found, if 2 then print all solutions (trivial variants are not included, nor are those which repeat any situation), if 3 then print the best (least number of boat trips) solution

BOATLOAD - the number of people the boat can carry

CREW - the number of rowers needed to man the boat

RM - number of missionaries who can row

RC - number of cannibals who can row

NRM - number of missionaries who cannot row

NRC - number of cannibals who cannot row

The various internal variables are as follows:

LEFT,RIGHT - vectors of integers representing the numbers of people on the left and right banks. Fixed size vectors are not used since SIDE must be able to point to them.

SIDE - points to either LEFT or RIGHT. It is backtrackable as are all changes to the elements of LEFT and RIGHT made through it.

LOAD - number of people to take on current boat trip

B1, B2, B3, B4 - numbers of rowing missionaries, rowing cannibals, non-rowing missionaries and non-rowing cannibals to take on current trip

B1A, B2A, B3A, B4A - as B1, B2, B3, B4 but refer to the previous trip. They are used to prevent the immediate undoing of what a trip has accomplished.

COMEBACK, GOOVER - lists of integers which will successively be tried as values for LOAD when the boat is coming back and going over

MOVE - indicates which of COMEBACK or GOOVER is currently being done

BANK - contains the string representing the destination bank
ACTIONS - a list of the solutions found, each of which is a list of strings representing the required actions
ACTS - contains the strings representing the actions done so far in the current solution. The elements of it are backtrackable.
BEST - temporary register used when determining the best solution
M, N - temporary context registers
C - points to the context in effect when MISS_CAN2 is called. Checking of situations should not go beyond this one, and on exit, it should be as it was and should be the current context.
CL - saves the original children of C so that on exit, only the extra ones created by MISS_CAN2 are destroyed
SAVE - saves the values of the two system flags altered by MISS_CAN2 so that they can be reset on exit

The procedures MIN and MAX find the minimum and maximum of the integers passed as arguments. (Note the use of multiple parameters.) CDESTROY destroys its argument context along with all of its descendents.

The following conversational version of the program is used to describe it:

```
If some parameter is bad, print an error message and
stop.
If the parameters are such that no solution is possible,
print a message to that effect and stop.
Set the system flags so that everything is not
automatically backtracked (MISS_CAN2 is clean and does
not need this) and so that each 'try' statement produces
a new context whenever executed.
The 'failing' statement will be backed up into (it is not
executed when encountered in the forward direction) only
when all solutions to this particular problem (if there
are any) have been found and put on ACTIONS. If and when
such a backup does occur then
   If no solutions were found, print a message to this
   effect or,
   if all solutions were requested (CODE=2) then print
   all of them or,
   find and print the best solution (if more than one is
   found with the same number of trips, then the first
   one found is printed).
   After the printing, a branch to EXIT is made to "clean
   up after the execution of the procedure" and stop.
The actual search for solutions now commences. The method
is to make boat trips WHILE there are still some people
remaining on the left bank. Each boat trip consists of:
   Empty the boat. (Internally, B1, B2, B3 and B4
   represent the contents of the boat.)
   Select some total number, LOAD, of people to put into
   the boat. When going over, large numbers are tried
```

first, when coming back, small numbers are tried first (a heuristic to aid the single solution case). If a fail back to this point occurs, then select another value for LOAD if there are any, else fail back some more. The values for LOAD tried must be less than or equal to the number of people on the side the boat is departing from.

Successively try values for B1, B3 and B2 (B4 is then determined exactly since B1+B2+B3+B4 = LOAD). The various constraints used ('from', 'to' and 'suchthat' parts) are described in the comments. It would have been simpler but less efficient to try all values and then later fail if they were bad.

If the values just selected are such that they undo what the previous boat trip accomplished, then fail (prevent simple looping).

Remove the selected people from the vector representing the departure side.

Cross to the other side.

If, when the boat gets there, the cannibals would eat the missionaries, then fail.

Add the selected people to the vector representing the arrival side.

The next nine lines of the program (starting with "M:=*MYCONTEXT") generate a fail if the situation just produced has ever existed before. The test is made by comparing the values of LEFT as they are now with their values in previous situations (WRT context M). If they are all the same (in any of the contexts M) then a fail is generated. The contexts M to test are every eighth one between the current one and C. Only every eighth one need be tested since each situation involves four 'try's, hence four contexts, and only every second situation has the boat on the same side.

Following this last test, the crossing is finalized (at least until a later fail) by updating the B1A - B4A 'previous trip' values and by putting appropriate messages onto ACTS, the list of steps for the solution.

The loop which makes a boat crossing is now repeated if a solution has not been created.

Execution will reach this point only if a solution has been found.

If more than one solution is required (CODE¬=1), then the found solution (ACTS) is added to the list of solutions (ACTIONS) and a fail is produced to generate more solutions (if any).

If only one solution was requested, it is printed.

EXIT: This point is reached when processing is complete, but unwanted contexts and backtrack points remain.

The extra backtrack points are removed ('succeed "point"').

The original context is restored.

The original global flag values are restored.

Each child of the current context which is not in CL (i.e. did not exist originally) is destroyed along with all of its descendents.
The children list of the current context is reset to its original value. (Parse time setting of the global *PERMIT flag is needed to get the parser to allow this tampering with things the system is dependent on.)

The following procedure, PARAND, takes as argument a set of boolean 'expr's and finds the logical AND of them. It does this in an unusual way however. Instead of evaluating the expressions serially and stopping when one yields false, the expressions are executed all at once, in parallel. Unless the expressions interact the result will be the same, but the time taken to obtain that result may differ drastically. If one of the expressions (it is not known which one or ones) yields a false after only a small amount of execution, then the entire procedure call will take only that time multiplied by the number of expressions plus some overhead for PARAND. If the expressions were evaluated serially, the time required would be the sum of the times for the expressions evaluated up to the one which yielded false. Which type of processing is preferable will depend on the particular situation. PARAND works by using a 'ptry' to produce a list of processes which have, essentially, the evaluation of one of the 'expr's as body. These processes are all started and the procedure waits for a termination which yields a false (or for all to finish, in which case the result is true). No new contexts are created (unless an 'expr' does so) and all processes used by PARAND are destroyed.

```
bool proc PARAND(bool expr set E);
   (bool F:=true, SAVE:=*PBTCONT;
   process P1,P:=*MYPROCESS;
   bool expr X;
   process list loc L;
   event_type EVEN;
   *PBTCONT:=false;
   L:=for X in E ptry
      resume P;
      F:=F&=X;
      suspend *MYPROCESS then;
   EVEN:=any inactive &L;
   *PBTCONT:=SAVE;
   for P1 in &L do resume P1 time 1 od;
   while F&for P1 in &L using bor collect P1@*STATUS>=0
      do ignore wait EVEN od;
   destroy EVEN;
   for P1 in &L do
      terminate P1;
      destroy P1 od;
   F);
```

The following procedure, NIM, is a complete program for playing the game of nim against a human opponent. The object is to remove the last counter from a number of piles of counters. The players alternate, removing at least one counter in each turn, but taking counters from only one pile. The strategy employed by the program is a purely calculated one based on the following procedure: let D be the bitwise exclusive-or of the binary representations of the numbers of counters remaining in the various piles; then, for each pile, let F be the result of subtracting D from the integer represented by the bitwise exclusive-or of D and the number of counters in the pile; if F is positive then removing that number of counters from that pile is a winning play. If no winning plays exist, then one counter is removed from the largest pile. The program is user proof in that no possible input can cause it to fail. The input is scanned via string pattern matching so that an illusion of "understanding" is produced.

```
proc NIM(ints PP);
   int T,C,P,M,LEFT:=for P in PP using iplus collect P,
      NUMPILES:=size PP;
   bool GAMEOVER:=NUMPILES<1|for P in PP using bor
      collect P<1;          bits D;
   pat INTP:=D(string('0'|'1'|'2'|'3'|'4'|'5'|'6'
      |'7'|'8'|'9')@);
   int list list F;
   int list MP,PILES:=for P in PP using listconc
      collect L<P>;
   string M,CS,PS;
   proc PLAY(int C,P);
      PILES(P):=PILES(P)-C;
      LEFT:=LEFT-C
   corp;
   if GAMEOVER then line("Invalid piles. Start over.") fi;
   *I_L:=-1;  /*use free format output */
   while ¬GAMEOVER
   do line("The piles are ",(repseq PILES)(2|*L-1),".");
C The machines's move:
      D:=for P in PILES using logxor collect %P;
      F:=for P in PILES
         for C
         suchthat M:=P-bitint(D*%P)>0
         using listconc
         collect L((C M));
      if F=null    /*if no winning play*/
      then T:=PILES(1)  /*find largest pile*/
         P:=1;
         for M in PILES
         for C    /*the pile number*/
         do if M>T then T:=M; P:=C fi
         od;
         C:=1
      else MP:=F(irand(size F));
```

```
         C:=MP(1); P:=MP(2)
      fi;    /*select a random winning play*/
      PLAY(C,P);
      line("I'll take ",C," from pile ",P,
         " leaving ",(repseq PILES)(2|*L-1),".");
      if LEFT=0    /*The machine has won*/
      then line("Better luck next time.");
         GAMEOVER:=true
      fi;
C The human's move:
      for M to 3
      while ¬GAMEOVER
      do line(case M of ("Your move.",
            "That's an illegal move. Please try again.",
            "Move should be 'c from p' where p is the pile"+
               " and c # counters you wish to take."));
         if D(string(break("resign")--))<>M:=getcard
         then line("That's too bad.");
            GAMEOVER:=true   /*human resigns*/
         elif D(string(to(INTP)=CS break(" from ")
               to(INTP)=PS --))<>M&
            (C:=intget(CS);
            P:=intget(PS);
            P>=1&P<=NUMPILES&C>=1&C<=PILES(P))
         then PLAY(C,P);
            M:=4;
            if LEFT=0    /*The human has won*/
            then line("Drat. You won.");
               GAMEOVER:=true
            fi
         elif M=3  /*Invalid input for the third time*/
         then line("Perhaps you should study the 'nim'",
               " writeup. We'll play again another time.");
            GAMEOVER:=true
         fi
      od
   od
corp;
C Sample session:
:NIM(3,4,5);
 The piles are (3 4 5).
 I'll take 2 from pile 1 leaving (1 4 5).
 Your move.
 I wish to take 5 counters from pile #3.
 The piles are (1 4 0).
 I'll take 3 from pile 2 leaving (1 1 0).
 Your move.
 damn
 That's an illegal move. Please try again.
 O.K. O.K. I'll take 1 from 2 you ?!##<!
 The piles are (1 0 0).
 I'll take 1 from pile 1 leaving (0 0 0).
 Better luck next time.
:
```